TITLE:

Index

SIZE A	DWG. NO. A1	REV
SCALE: 1:4		SHEET 1 OF 1

Name	Part Number	Number	Price Per	Vendor
12"x24" 22 Guage Steel Sheet	N/A	1	$14.38	Home Depot
1" Corner Brace	N/A	12	$0.6575	Home Depot
Sleeve Bearing	6723K11	4	$6.77	McMaster Carr
M4x10 Screws (100)	92095A476	1	$8.04	McMaster Carr
M4 Locknut (100)	94645A101	1	$14.05	McMaster Carr
1.5" L Bracket	1556A63	4	$0.81	McMaster Carr
Motor Mount (ACC Conduit Hanger)	N/A	4	$0.95	Home Depot
Motor	AM-4230	2	$45.00	Andy Mark
Motor Cushion	HC40C72P193	1	$1.95	Home Depot
M6x10 Screw (100)	92095A234	1	$13.68	McMaster Carr
M6 Locknut (100)	94645A205	1	$13.39	McMaster Carr
Aluminum Composite Material 36"x48"x1/8"	N/A	1	$65.88	Home Depot
Tetrix 4mm Bushing (12)	W41792	1	$15.95	Pitsco
10mm Bore Axle Collar	G0318906	16	$2.73	Zoro
60mm Standoff		10		Amazon
Linear Slide	6723K2	2	$17.50	McMaster Carr
Tetrix Axle (6)	W39088	2	$17.95	Pitsco
Tetrix Axle Collar (6)	W39092	3	$4.50	Pitsco
1/2" Axle		1		Amazon
Roller Bearing	25015T24	10	$7.42	McMaster Carr
1/2" Bore Axle Collar (2)	AZSSMUK	6	$9.99	Amazon
BRECOFlex Tread (2)	50 TK5K6/1100 V	1	$188.94	BRECOFlex

TITLE:

Bill of Materials

SIZE A | DWG. NO. A2 | REV

SCALE: 1:4 | SHEET 1 OF 2

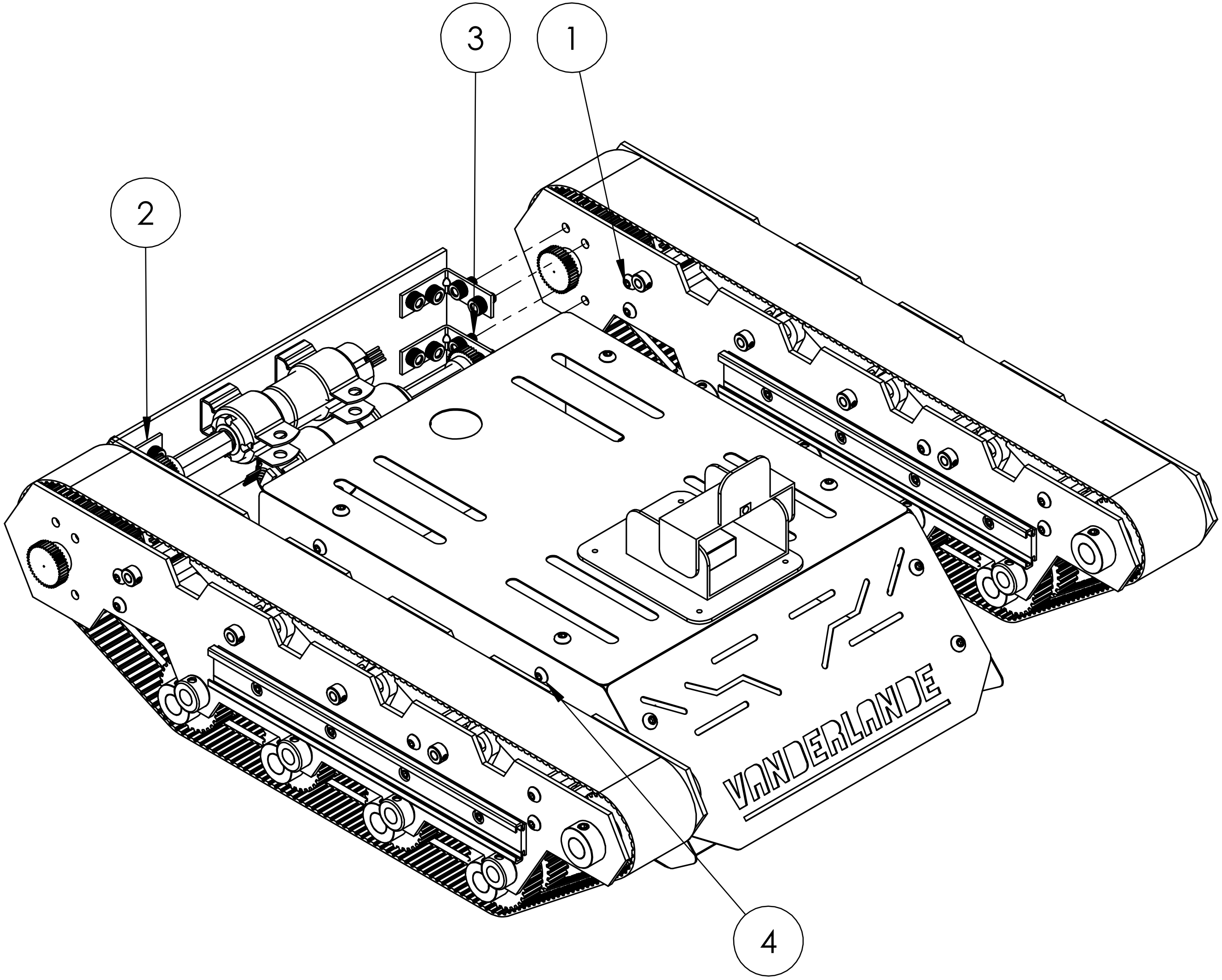
Name	Part Number	Number	Price Per	Vendor
Aluminum Sheet, HAKZEON 6"x12"x1/4"	N/A	1	$40.99	Amazon
RC Spring Shock (Yeah Racing 90mm Two Stage Internal Spring Shock) (2)	N/A	4	$27.99	Amazon
Tetrix Spacer 1/8" (12)	W39100	2	$2.15	Pitsco
M4x20 Screw (50)	92095A196	1	$8.69	McMaster Carr
10mm Steel Axle	N179-804 4005BC	4	$15.78	Amazon

TITLE:
Bill of Materials

SIZE A DWG. NO. A2 REV

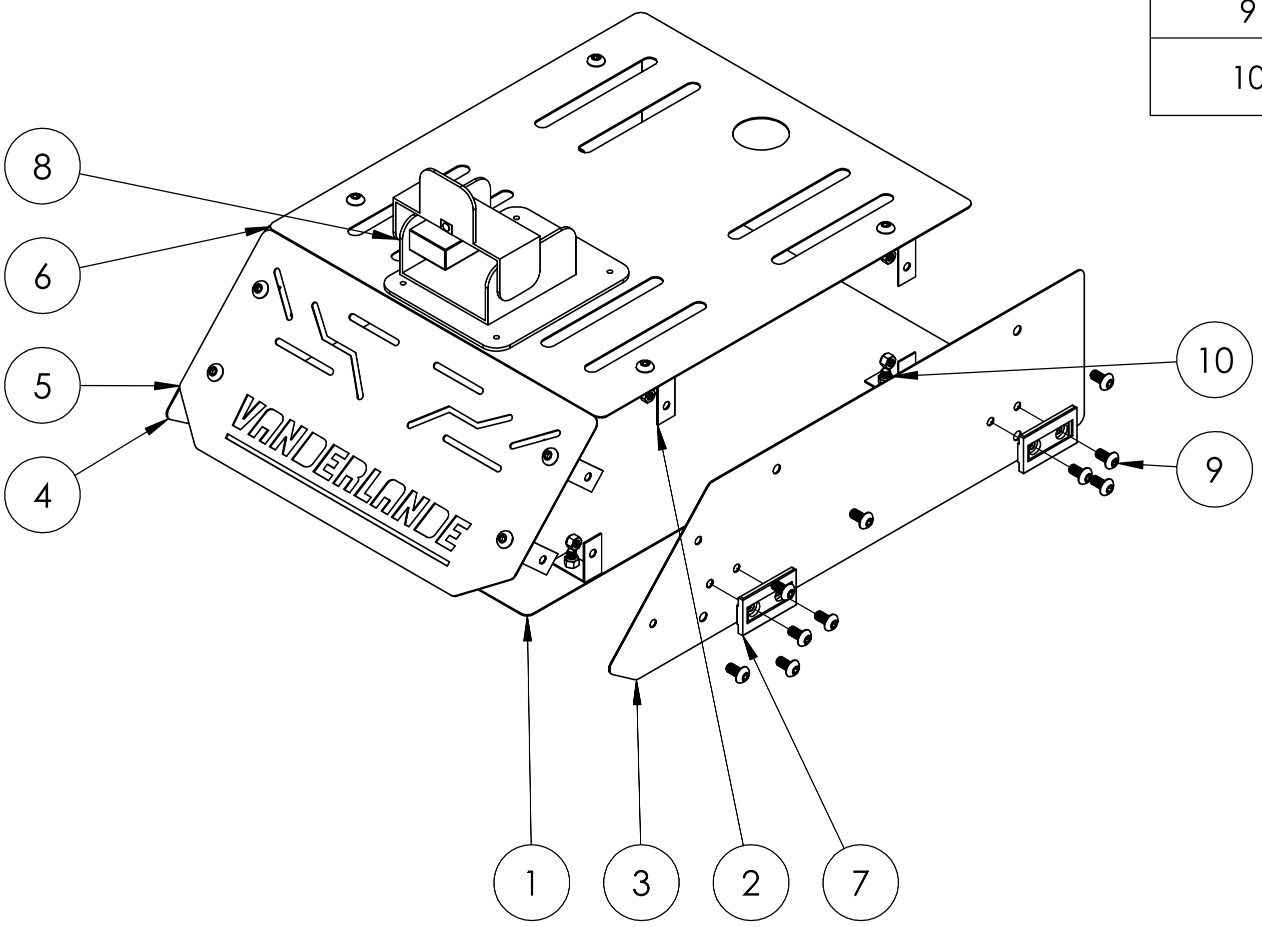
SCALE: 1:4 SHEET 2 OF 2

ITEM NO.	PART NUMBER	QTY.
1	Chassis Assembly	2
2	Motor Assembly	1
3	M6x10 Screws	8
4	Body Assembly	1



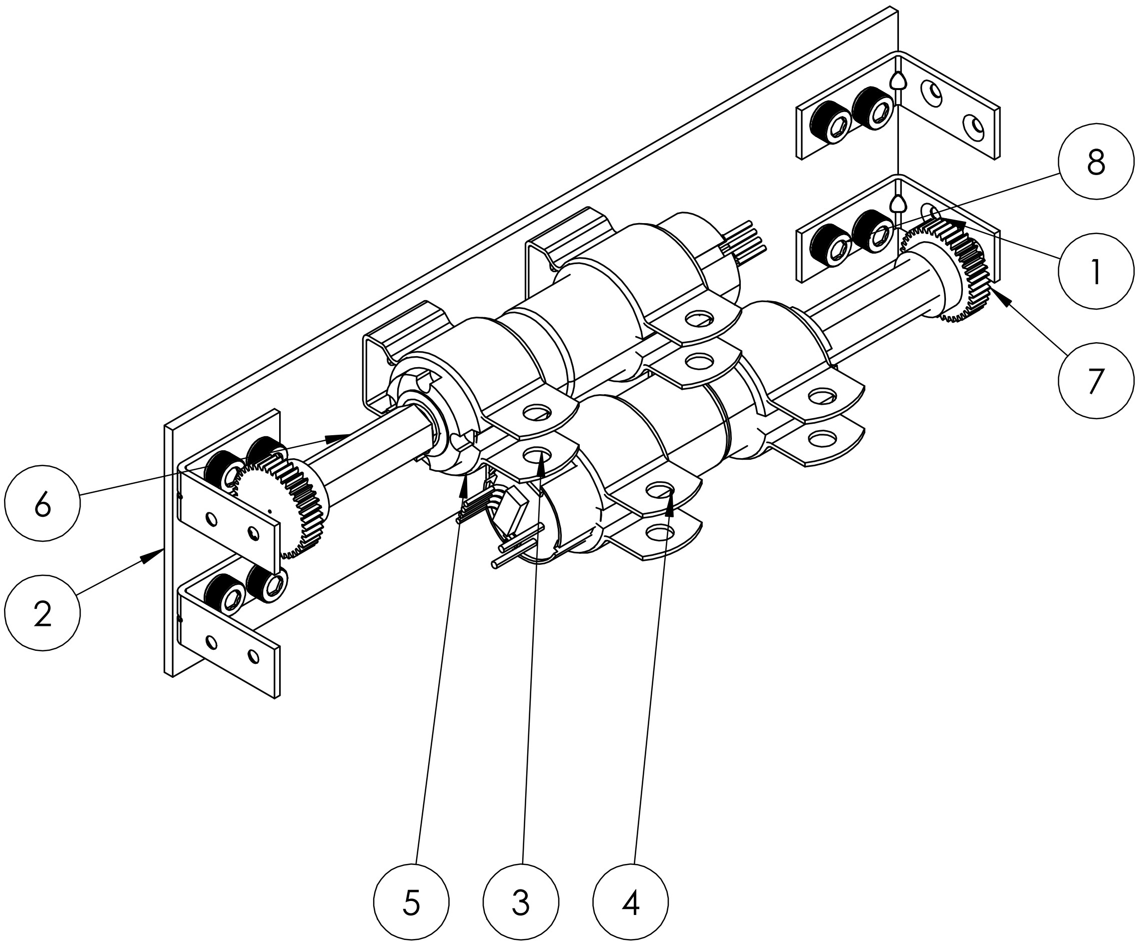
TITLE: Dia-Bot Assembly		
SIZE A	DWG. NO. 1	REV
SCALE: 1:4		SHEET 1 OF 1

ITEM NO.	PART NUMBER	QTY.
1	Bottom Panel	1
2	Bracket	12
3	LeftPanel	1
4	RightPanel	1
5	FrontPanel	1
6	TopPanel	1
7	Sleeve Bearing	4
8	Camera Mount	1
9	M4x10 Screw	30
10	M4 Locknut	30



TITLE:		
Body Assembly		
SIZE A	DWG. NO. 2	REV
SCALE: 1:4		SHEET 1 OF 1

ITEM NO.	Description	QTY.
1	L Bracket	4
2	Motor Mounting Plate	1
3	Motor Mount	4
4	Motor	2
5	Motor Cushion	4
6	Motor Output Shaft	2
7	Motor Gear	2
8	M6x10 Screw	8
9	M6 Locknut	8

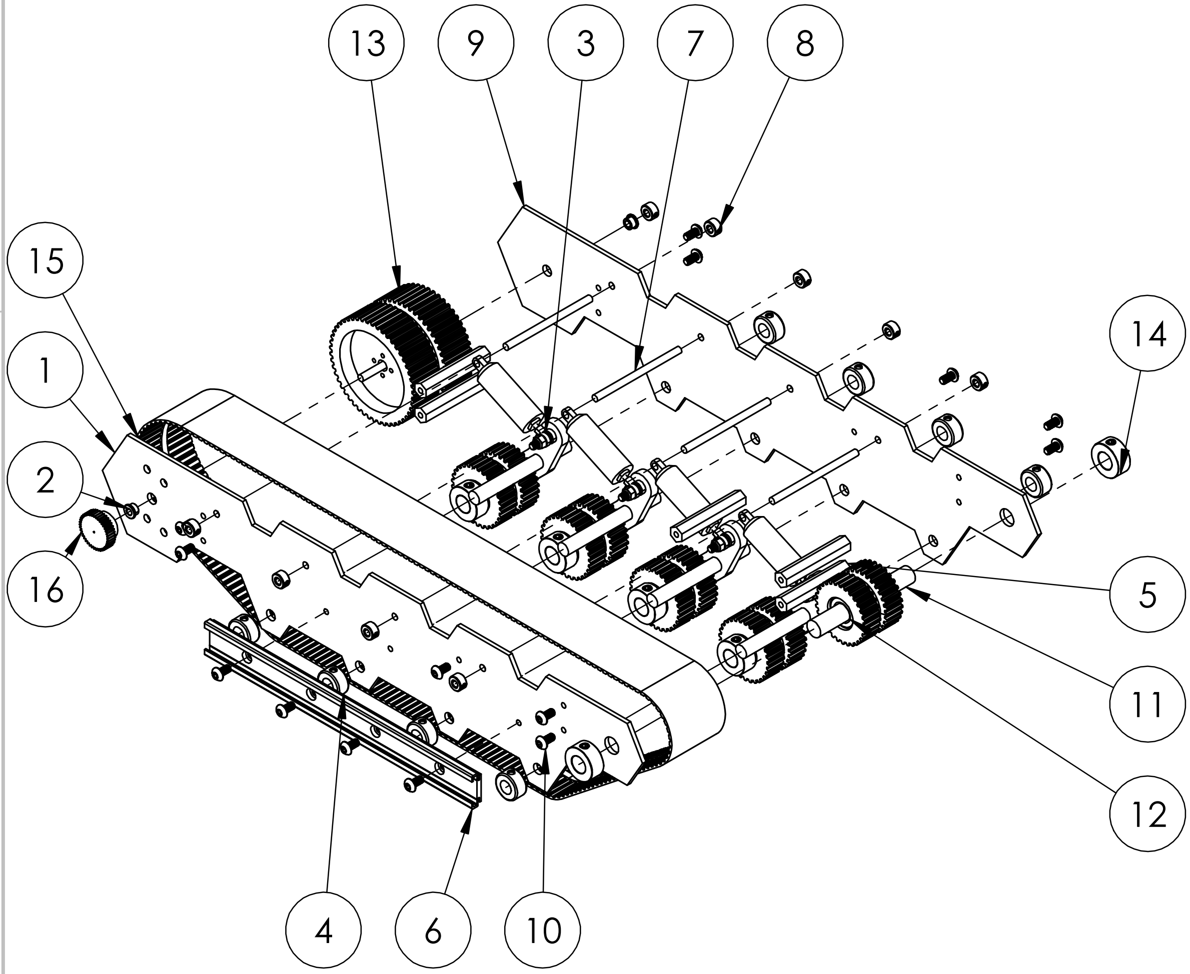


TITLE: Motor Assembly

SIZE	DWG. NO.	REV
A	3	

SCALE: 1:2

SHEET 1 OF 1



ITEM NO.	Description	QTY.
1	Interior Baseplate	1
2	Tetrix Bushing	2
3	Suspension Subsystem Assembly	4
4	10mm Bore Axle Collar	8
5	60mm Standoff	5
6	Linear Slide	1
7	Tetrix Axle	5
8	Tetrix Axle Collar	9
9	Exterior Baseplate	1
10	M4x10 Screw	14
11	1/2" Axle	1
12	Pulley and Roller Bearing	1
13	Drive Pulley	1
14	1/2" Bore Axle Collar	2
15	BRECOFlex Tread	1
16	Motor Gear	1

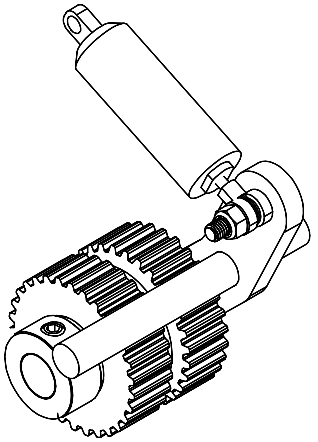
TITLE:

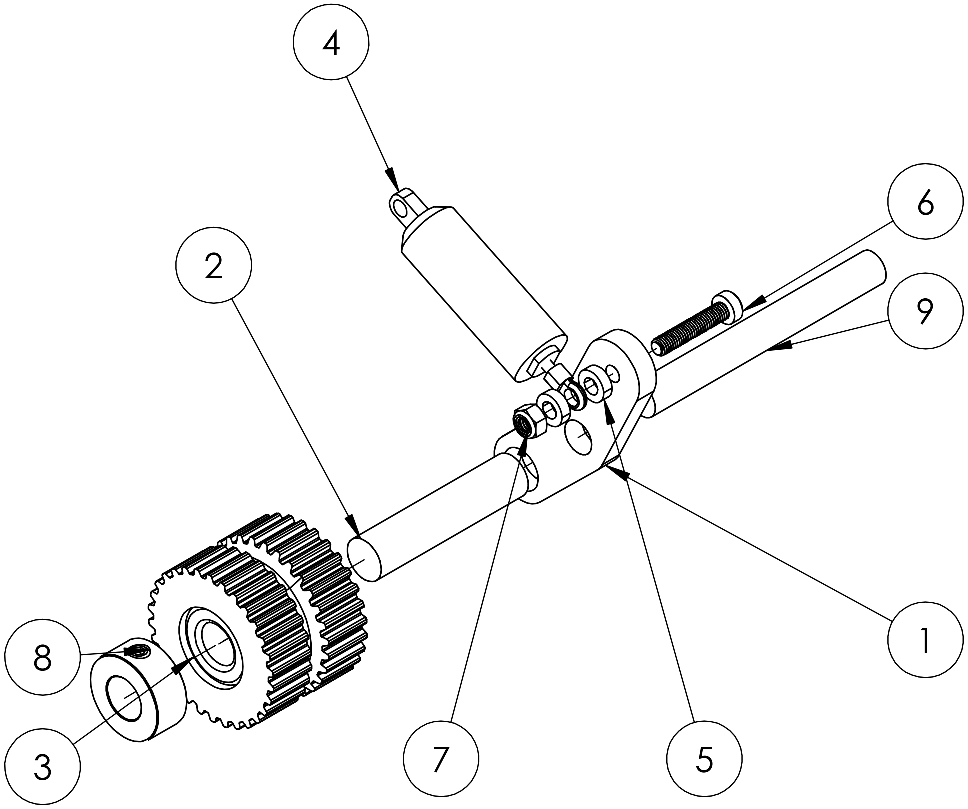
Chassis Assembly

SIZE A DWG. NO. 4 REV

SCALE: 1:10 SHEET 1 OF 1

ITEM NO.	Part Name	QTY.
1	Suspension Bar	1
2	1/2" Steel Axle	1
3	Pulley and Roller Bearing	1
4	RC Spring Shock	1
5	Tetrix Spacer	2
6	M4x20 Screw	1
7	M4 Locknut	1
8	1/2" Bore Axle Collar	1
9	10mm Steel Axle	1

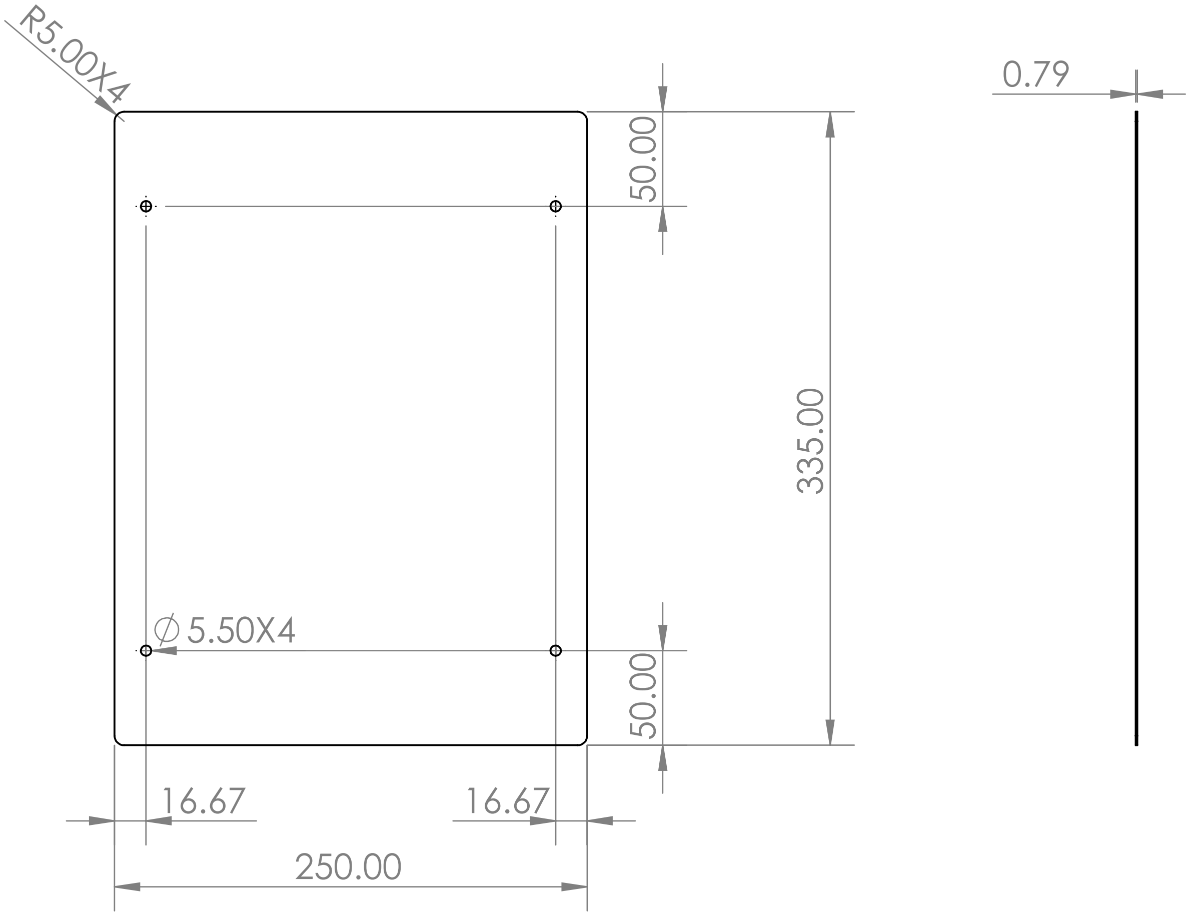




TITLE:

Suspension Subsystem Assembly

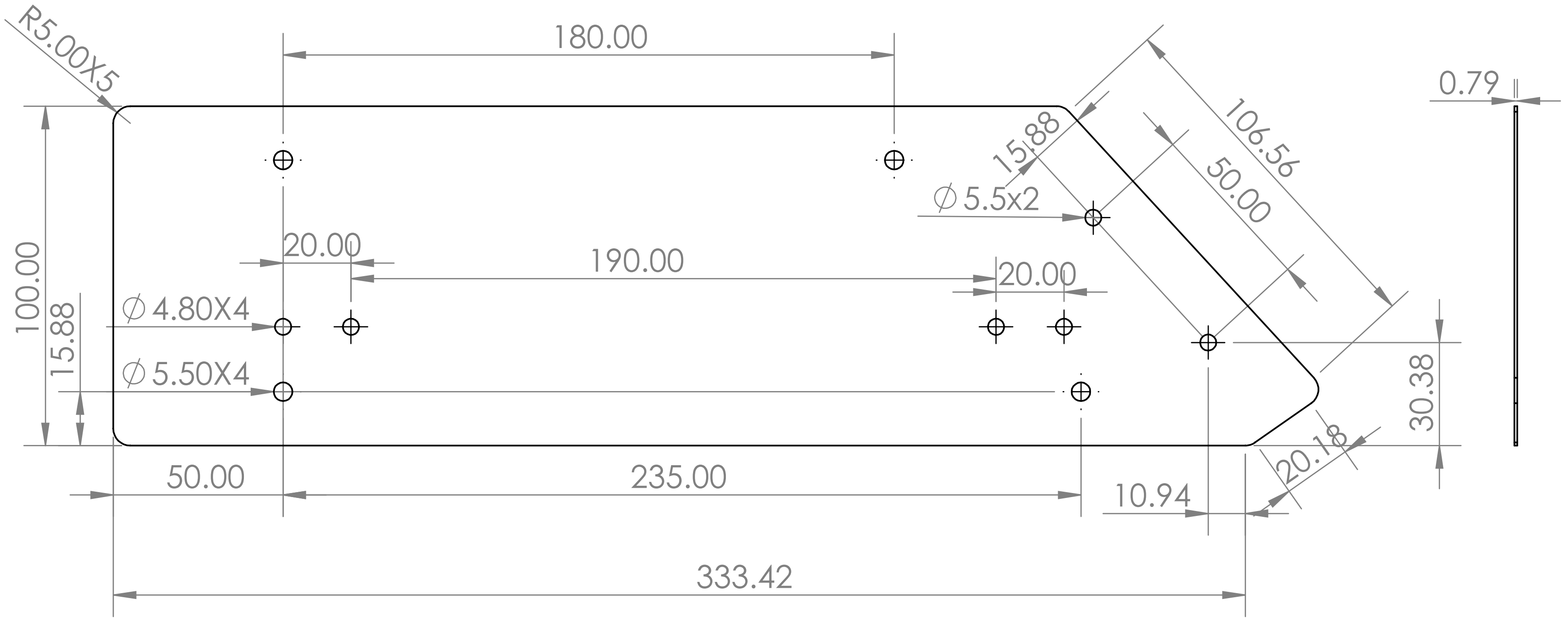
SIZE A	DWG. NO. 5	REV
SCALE: 1:5		SHEET 1 OF 1



TITLE:

Bottom Panel

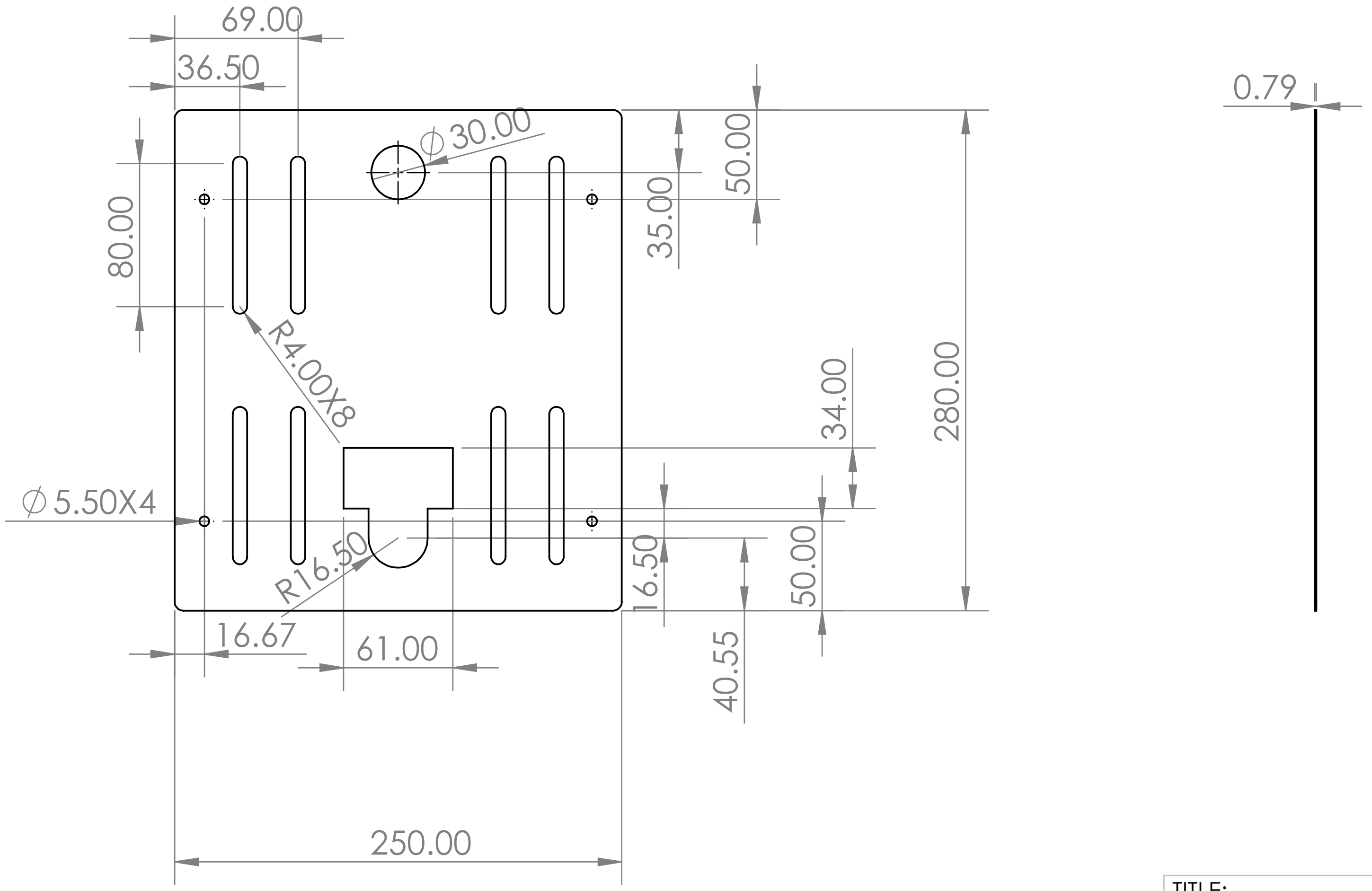
SIZE	DWG. NO.	REV
A	6	
SCALE: 1:4	Doug Walker	SHEET 1 OF 1



TITLE:

Side Panel

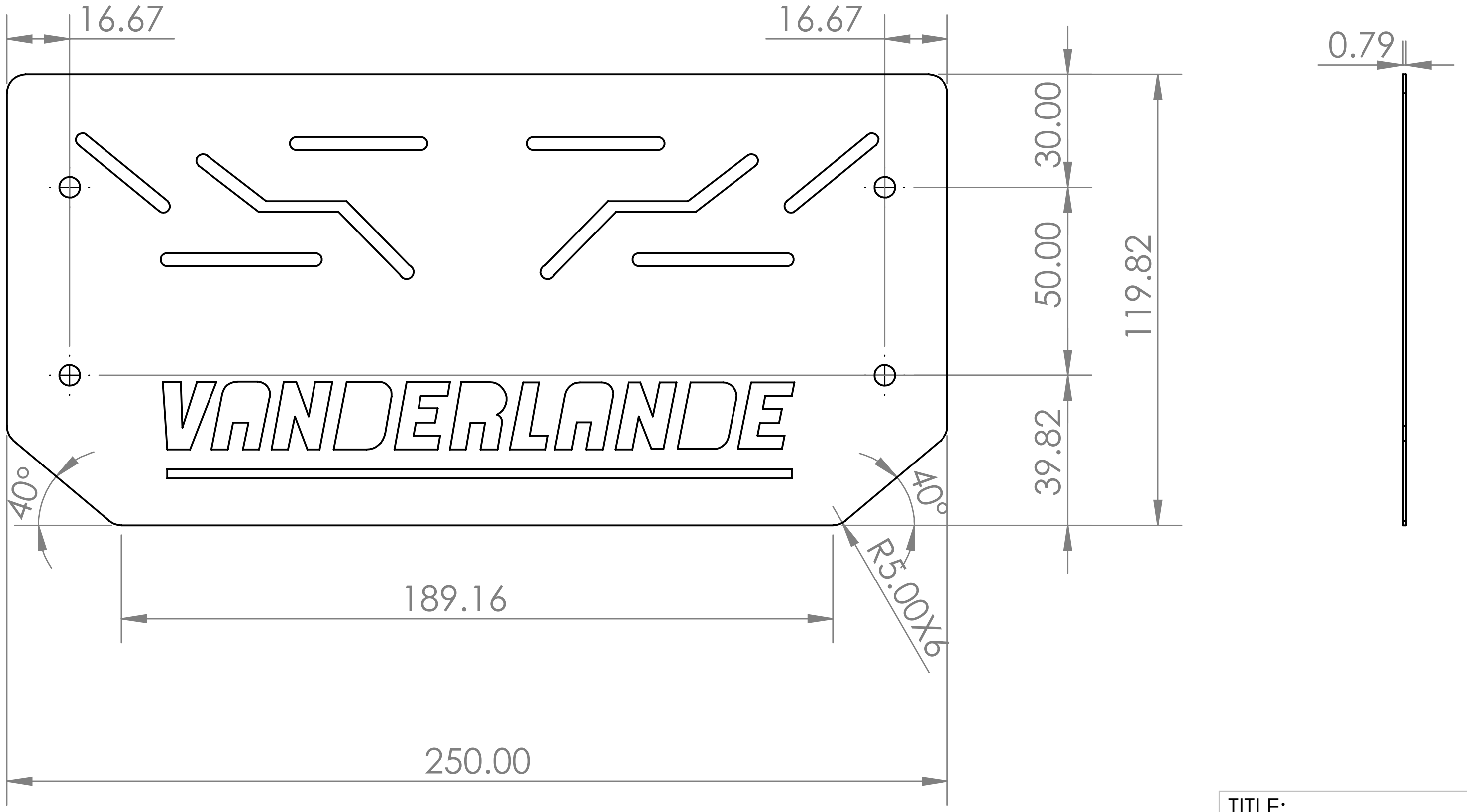
SIZE	DWG. NO.	REV
A	7	
SCALE: 1:2	Doug Walker	SHEET 1 OF 1



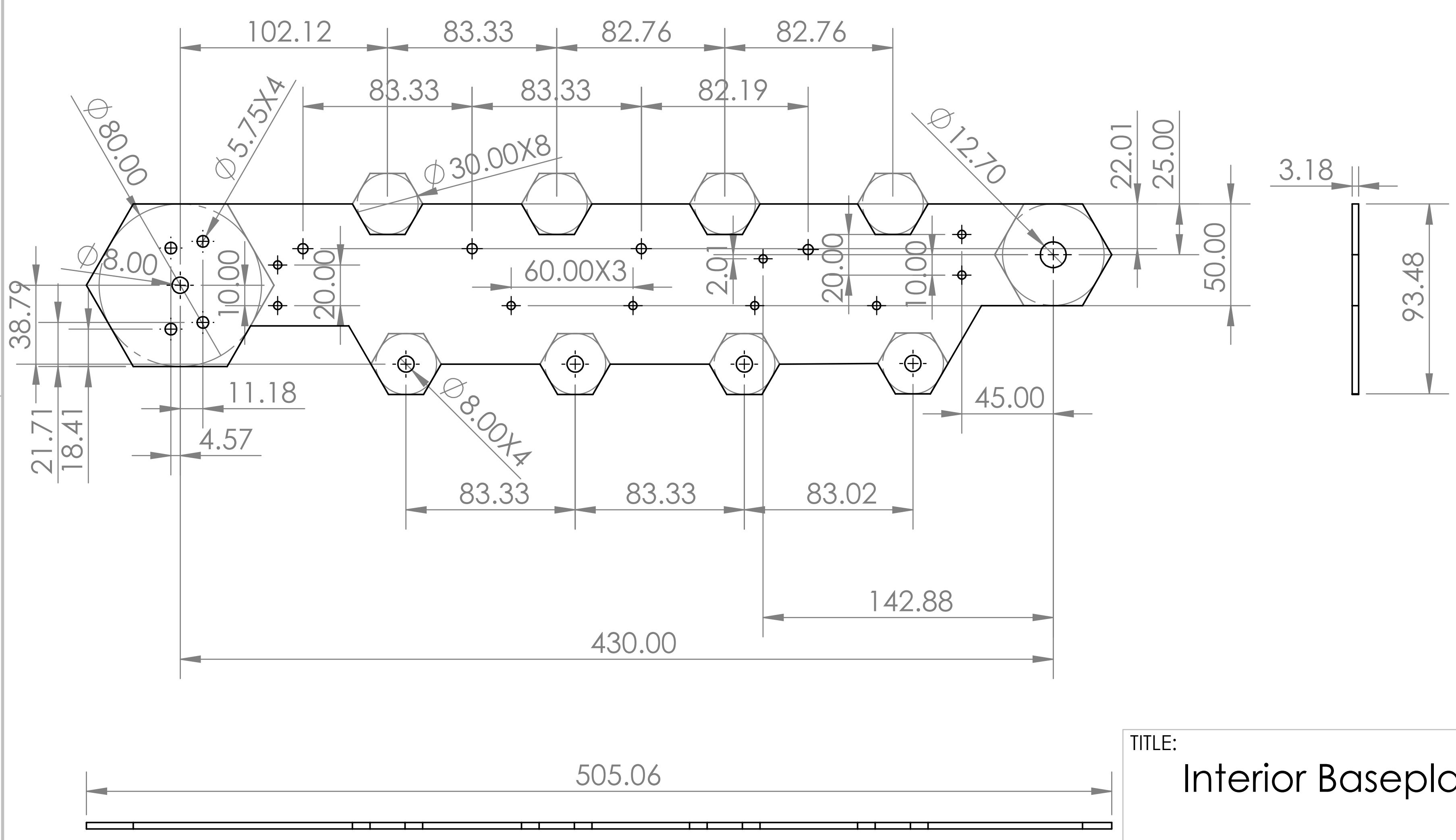
TITLE:

Top Panel

SIZE **A**	DWG. NO. 8	REV
SCALE: 1:4	Doug Walker	SHEET 1 OF 1



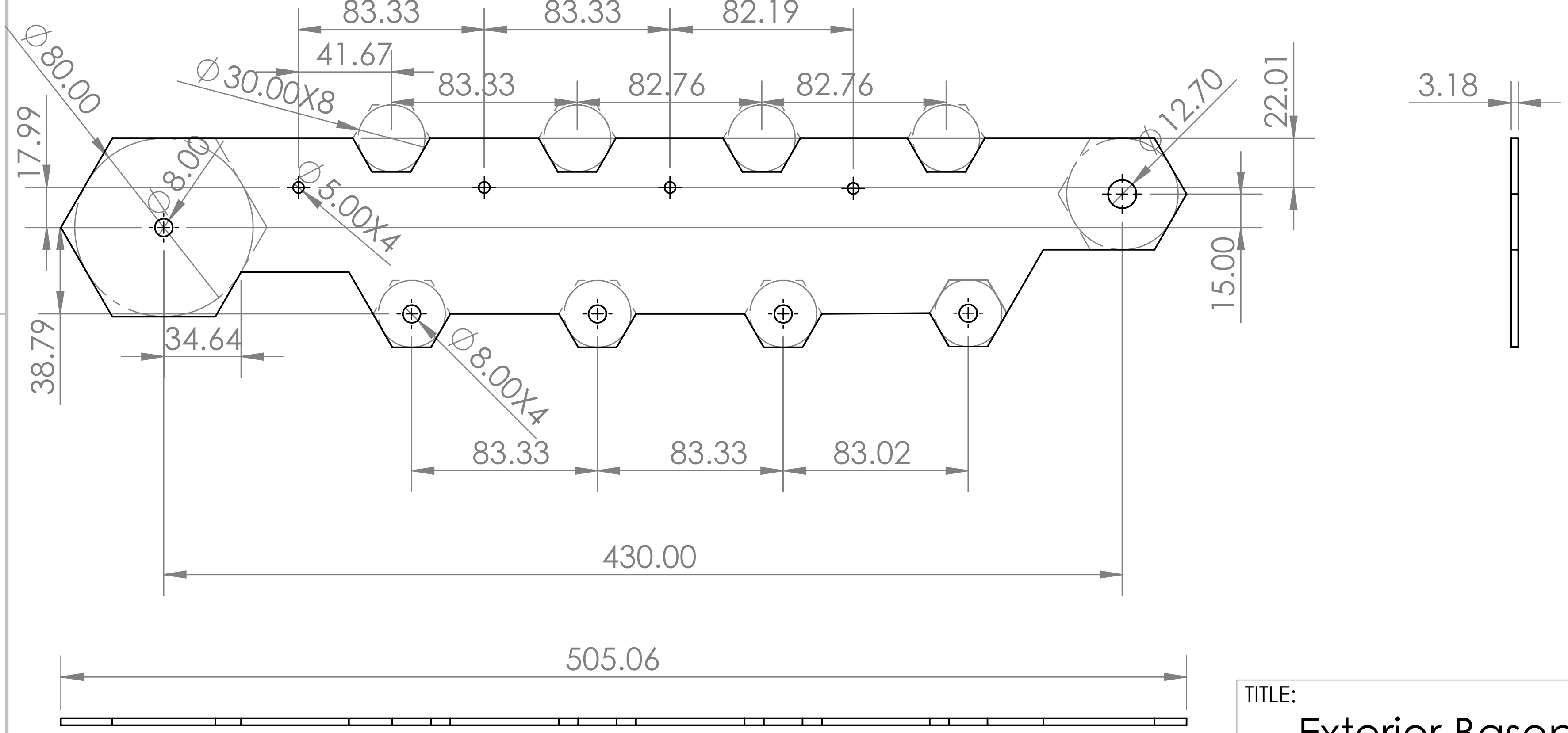
TITLE: Front Panel		
SIZE **A**	DWG. NO. 9	REV
SCALE: 1:2	Doug Walker	SHEET 1 OF 1



TITLE:

Interior Baseplate

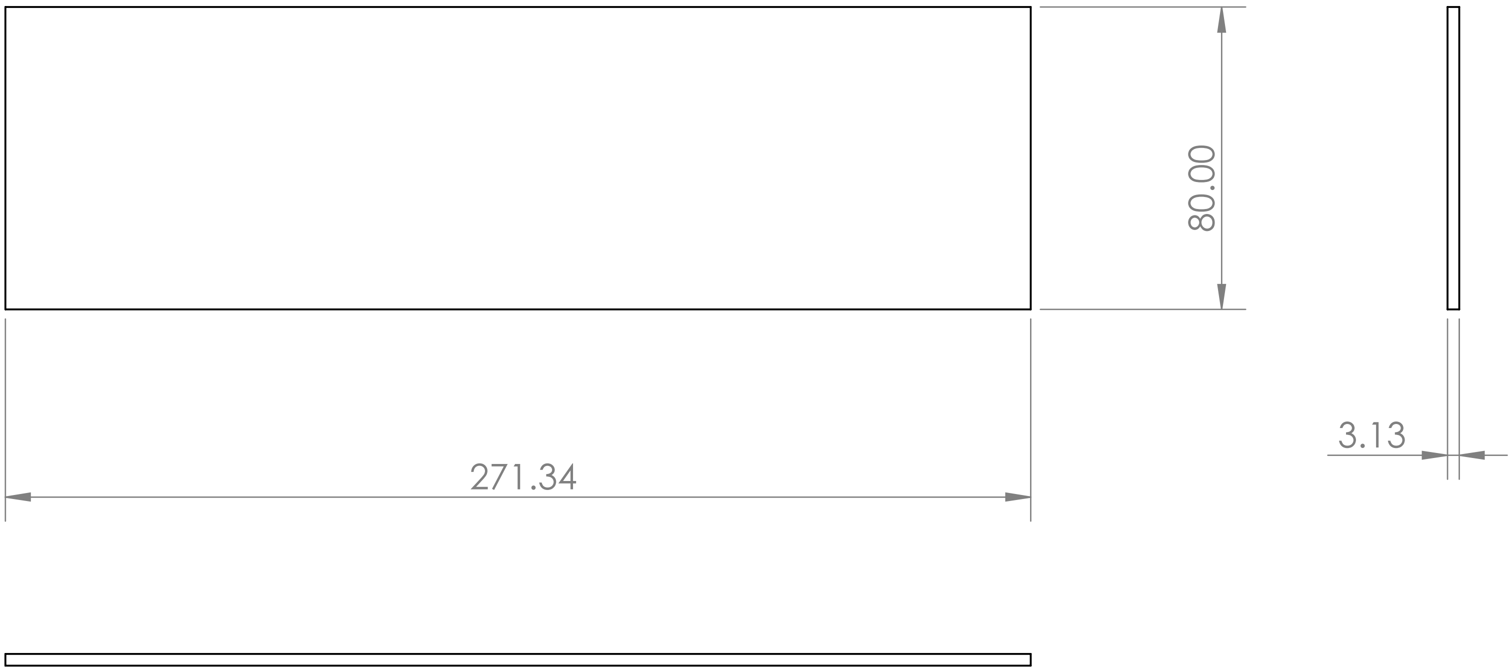
SIZE	DWG. NO.	REV
A	10	
SCALE: 1:3	Doug Walker	SHEET 1 OF 1



TITLE:

Exterior Baseplate

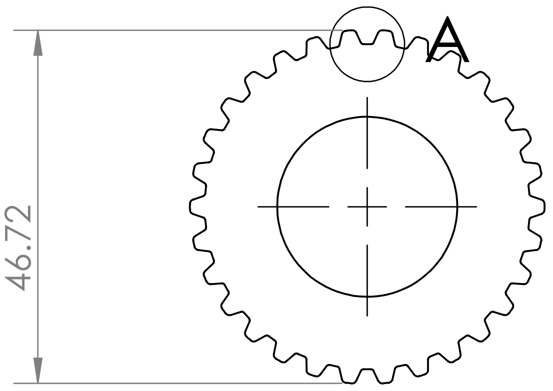
SIZE	DWG. NO.	REV
A	11	
SCALE: 1:3	Doug Walker	SHEET 1 OF 1

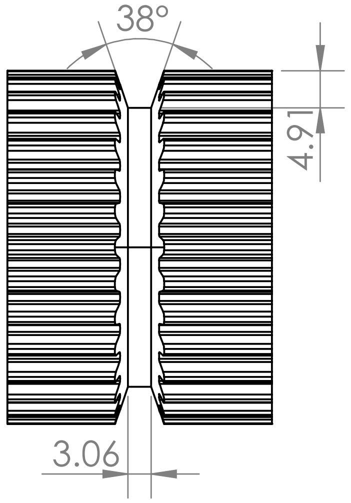


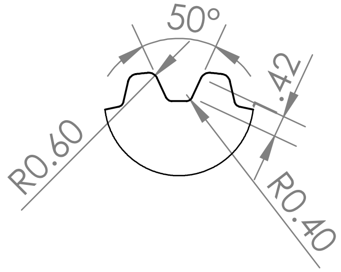
TITLE:

Motor Mounting Plate

SIZE A	DWG. NO. 12	REV
SCALE: 1:2	Doug Walker	SHEET 1 OF 1

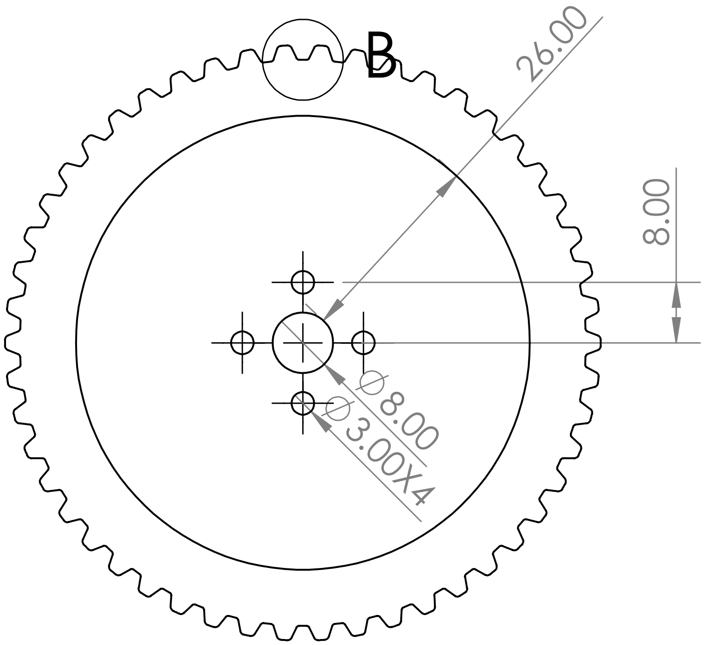


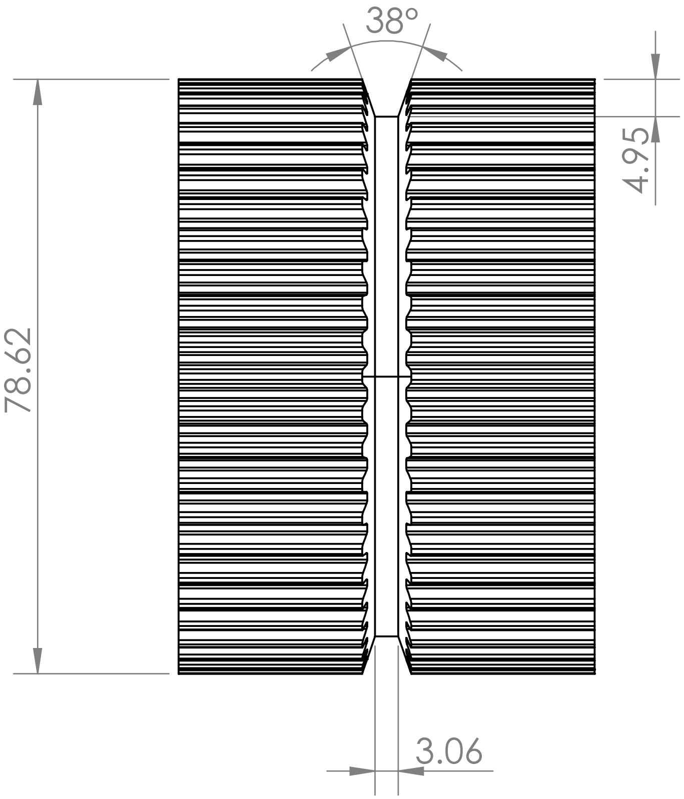


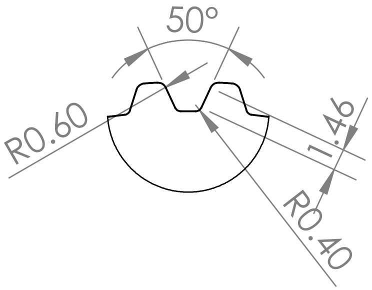


DETAIL A
SCALE 2 : 1

TITLE: Drive Pulley		
SIZE A	DWG. NO. 13	REV
SCALE: 1:1	Doug Walker	SHEET 1 OF 1



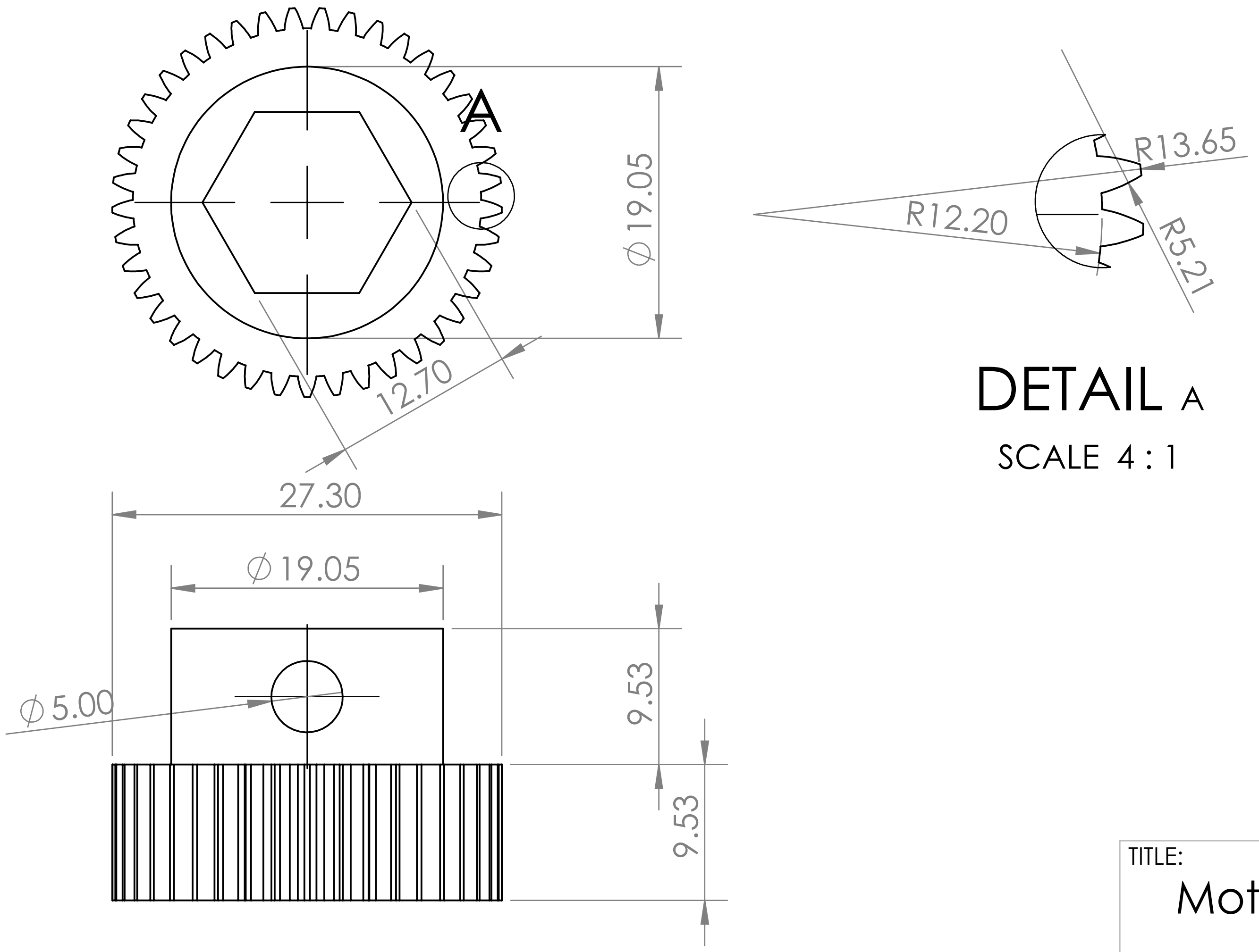




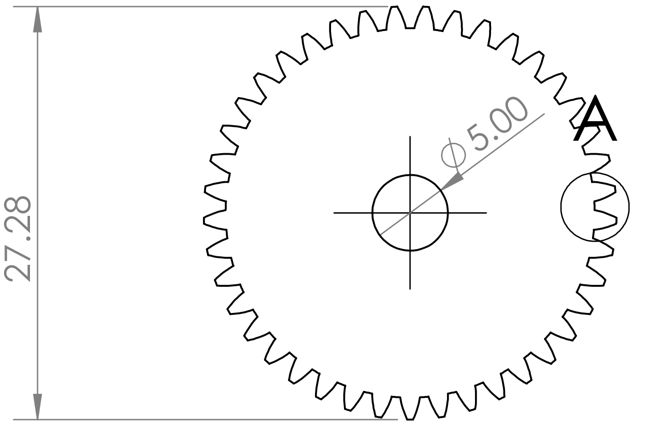
DETAIL B

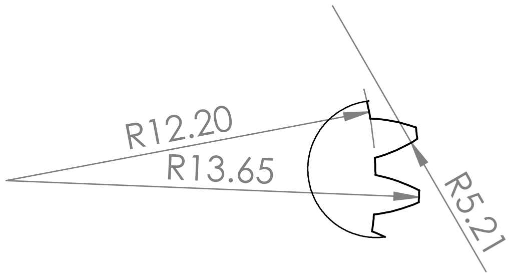
SCALE 2 : 1

TITLE: Idle Pulley		
SIZE A	DWG. NO. 14	REV
SCALE: 1:1	Doug Walker	SHEET 1 OF 1



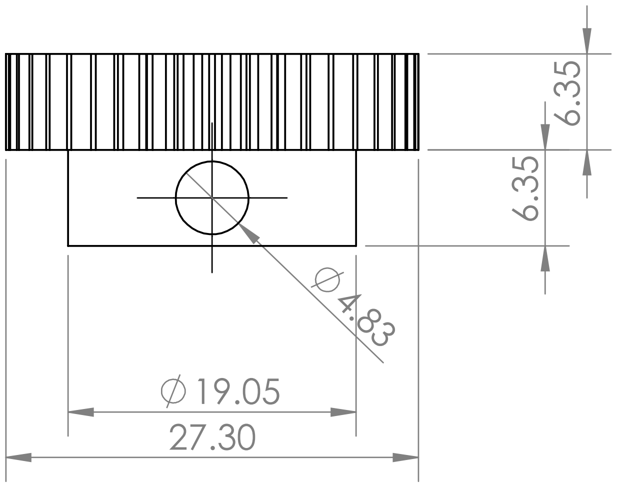
TITLE: Motor Driven Gear		
SIZE **A**	DWG. NO. 15	REV
SCALE: 2:1	Doug Walker	SHEET 1 OF 1





DETAIL A

SCALE 4 : 1

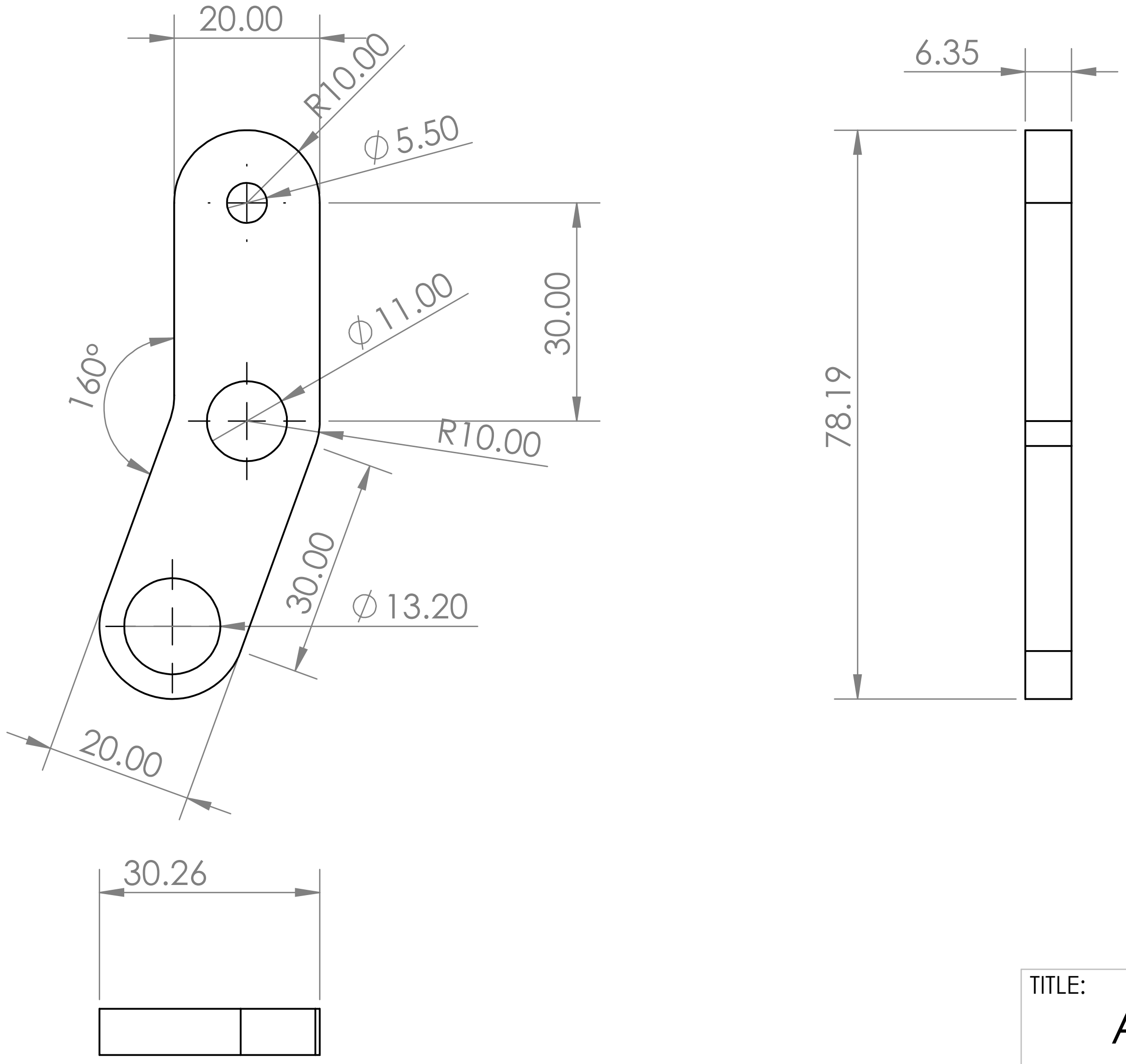


TITLE:

Sprocket Driving Gear

SIZE	DWG. NO.	REV
A	16	

SCALE: 2:1	Doug Walker	SHEET 1 OF 1



TITLE:

Aluminum Plate

SIZE	DWG. NO.	REV
A	17	

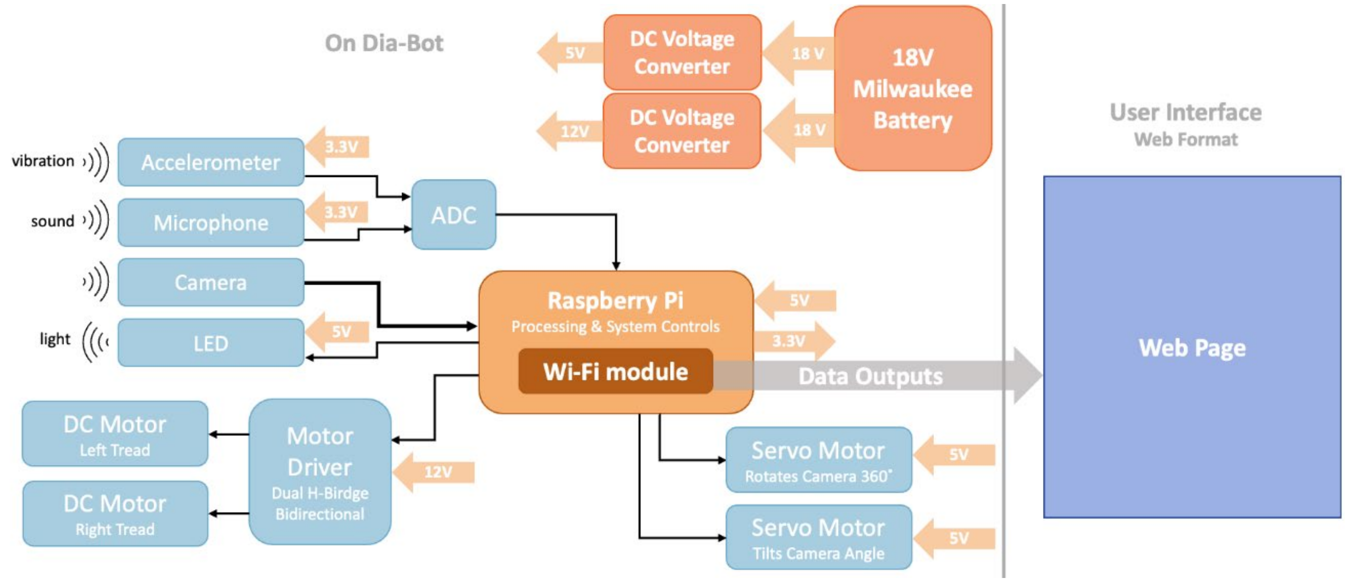
SCALE: 1:1	Doug Walker	SHEET 1 OF 1

Bill of Materials – Electrical

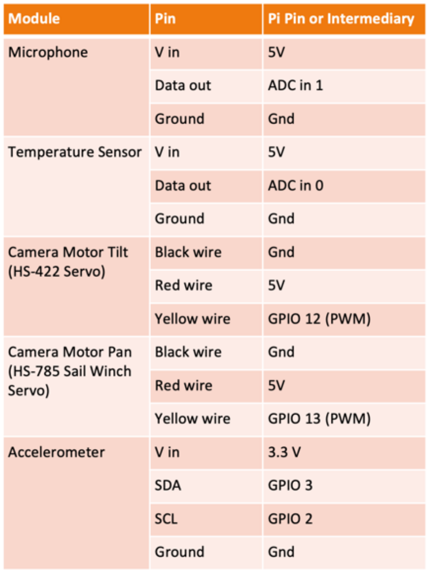
Dia-Bot Electrical Bill of Materials				
Name	**Part Number**	**Cost (Per)**	**Quantity**	**Link**
Raspberry Pi 4	4295 from Adafruit	$ 35.00	1	Buy a Raspberry Pi 4 Model B – Raspberry Pi
Raspberry Pi Spy Camera	1937 from Adafruit	$ 39.95	1	Spy Camera for Raspberry Pi : ID 1937 : $39.95 : Adafruit Industries, Unique & fun DIY electronics and kits
Camera Tilt-Pan System	WC-003-300 from SuperDroid	$ 218.58	1	Camera 360 Pan and Tilt System - Standard (superdroidrobots.com)
DC Motors (Drive)	AM-4230	$ 45.00	2	https://www.andymark.com/products/johnson-electric-gearmotor-and-output-shaft
Accelerometer	13963 from SparkFun	$ 4.00	1	SparkFun Triple Axis Accelerometer Breakout - LIS3DH - SEN-13963 - SparkFun Electronics
H-Bridge Driver	L9110H	$ 1.50	2	https://www.adafruit.com/product/4489
H-Bridge Breakout	TB6612	$ 4.95	1	https://www.adafruit.com/product/2448
Temperature sensor	MAX31820	$ 1.95	1	https://www.sparkfun.com/products/14049
Microphone (Electret)	MAX4466	$ 6.95	1	https://www.sparkfun.com/products/12758
Buck Converter: 18V-5V	MPM3610	$ 9.95	1	https://www.sparkfun.com/products/18375
Buck Converter: 18V-12V	WG8-40S1203	$ 15.99	1	Amazon.com: DC Voltage Reducer Converter DC 8V-40V to 12V 3A 36W Automatic Step Down Up Voltage Regulator Power Converter Waterproof Module Transformer for Golf Cart Club Car : Electronics
Milwaukee 18V Battery	M18	$ 30.99	1	For Milwaukee 18V Battery Replacement | M18 3.0Ah Li-Ion Battery — Vanon-Batteries-Store (vanonbatteries.com)
LED Ring	1643 from Adafruit	$ 7.50	1	NeoPixel Ring - 12 x 5050 RGB LED with Integrated Drivers : ID 1643 : $7.50 : Adafruit Industries, Unique & fun DIY electronics and kits
Total:		$ 468.81		

Electrical Connection Diagrams

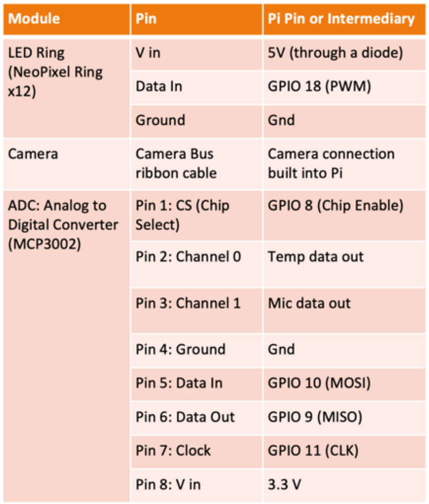
High-level electrical block diagram:



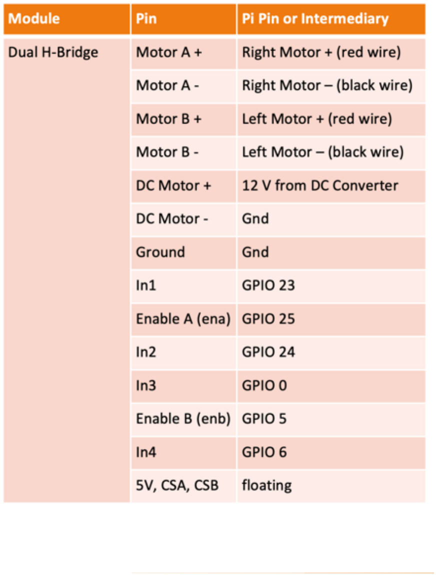
Pin assignments by module:



Module	Pin	Pi Pin or Intermediary
Microphone	V in	5V
	Data out	ADC in 1
	Ground	Gnd
Temperature Sensor	V in	5V
	Data out	ADC in 0
	Ground	Gnd
Camera Motor Tilt (HS-422 Servo)	Black wire	Gnd
	Red wire	5V
	Yellow wire	GPIO 12 (PWM)
Camera Motor Pan (HS-785 Sail Winch Servo)	Black wire	Gnd
	Red wire	5V
	Yellow wire	GPIO 13 (PWM)
Accelerometer	V in	3.3 V
	SDA	GPIO 3
	SCL	GPIO 2
	Ground	Gnd

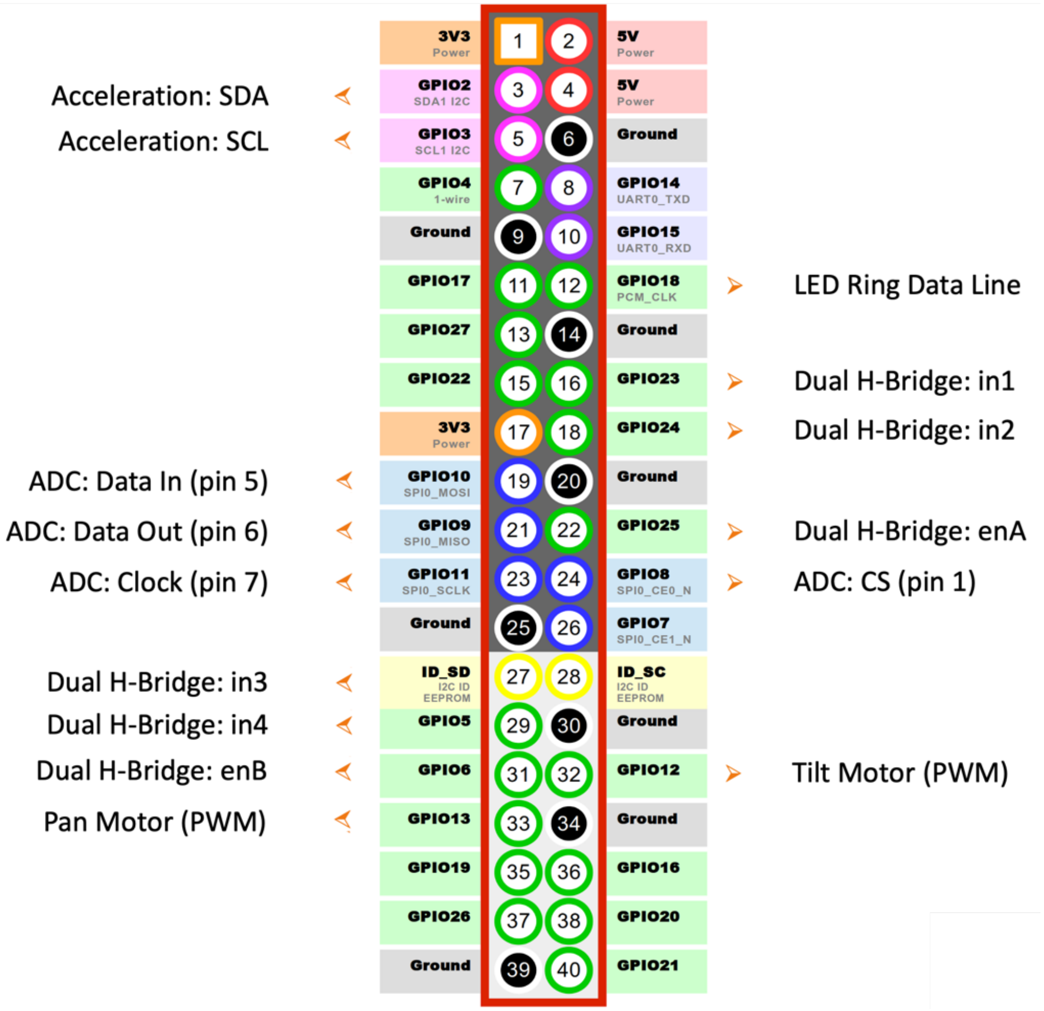


Module	Pin	Pi Pin or Intermediary
LED Ring (NeoPixel Ring x12)	V in	5V (through a diode)
	Data In	GPIO 18 (PWM)
	Ground	Gnd
Camera	Camera Bus ribbon cable	Camera connection built into Pi
ADC: Analog to Digital Converter (MCP3002)	Pin 1: CS (Chip Select)	GPIO 8 (Chip Enable)
	Pin 2: Channel 0	Temp data out
	Pin 3: Channel 1	Mic data out
	Pin 4: Ground	Gnd
	Pin 5: Data In	GPIO 10 (MOSI)
	Pin 6: Data Out	GPIO 9 (MISO)
	Pin 7: Clock	GPIO 11 (CLK)
	Pin 8: V in	3.3 V



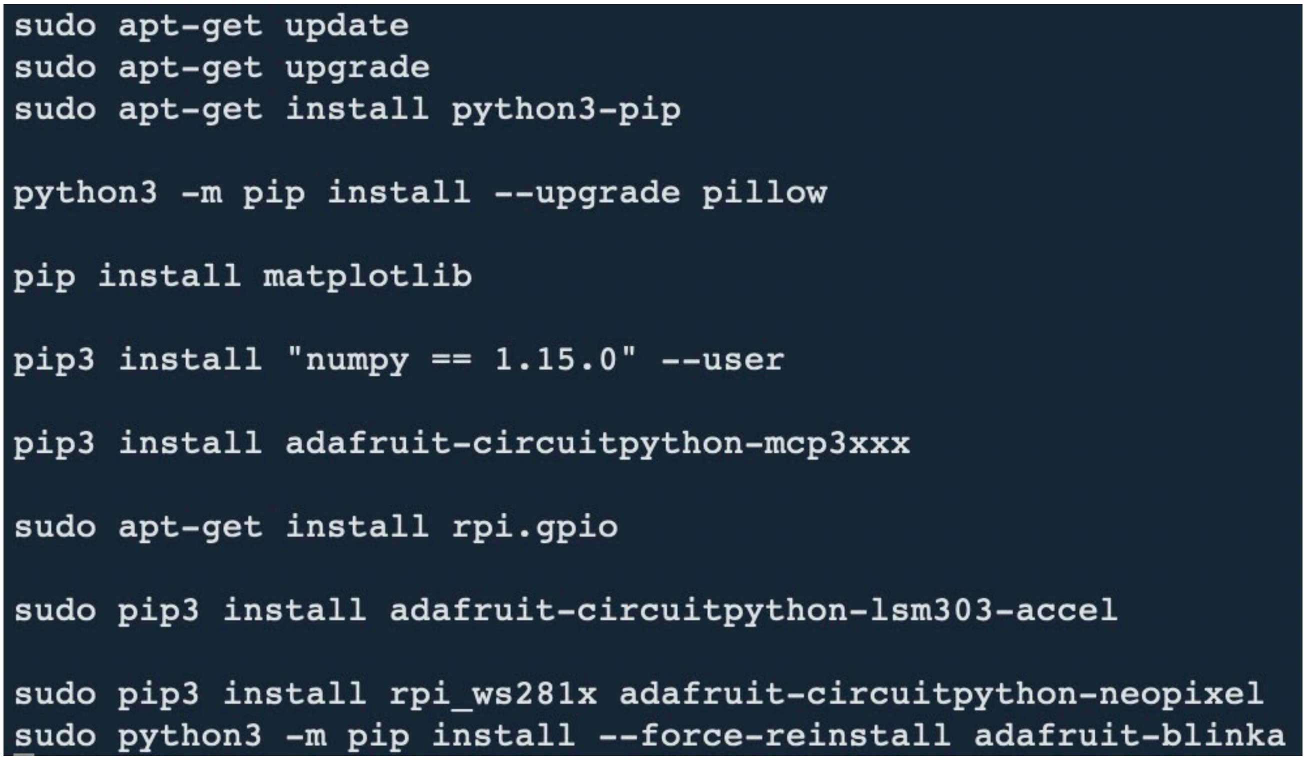
Module	Pin	Pi Pin or Intermediary
Dual H-Bridge	Motor A +	Right Motor + (red wire)
	Motor A -	Right Motor – (black wire)
	Motor B +	Left Motor + (red wire)
	Motor B -	Left Motor – (black wire)
	DC Motor +	12 V from DC Converter
	DC Motor -	Gnd
	Ground	Gnd
	In1	GPIO 23
	Enable A (ena)	GPIO 25
	In2	GPIO 24
	In3	GPIO 0
	Enable B (enb)	GPIO 5
	In4	GPIO 6
	5V, CSA, CSB	floating

Raspberry Pi 4 pin assignments:



	3V3 Power	1	2	5V Power	
Acceleration: SDA	GPIO2 SDA1 I2C	3	4	5V Power	
Acceleration: SCL	GPIO3 SCL1 I2C	5	6	Ground	
	GPIO4 1-wire	7	8	GPIO14 UART0_TXD	
	Ground	9	10	GPIO15 UART0_RXD	
	GPIO17	11	12	GPIO18 PCM_CLK	LED Ring Data Line
	GPIO27	13	14	Ground	
	GPIO22	15	16	GPIO23	Dual H-Bridge: in1
	3V3 Power	17	18	GPIO24	Dual H-Bridge: in2
ADC: Data In (pin 5)	GPIO10 SPI0_MOSI	19	20	Ground	
ADC: Data Out (pin 6)	GPIO9 SPI0_MISO	21	22	GPIO25	Dual H-Bridge: enA
ADC: Clock (pin 7)	GPIO11 SPI0_SCLK	23	24	GPIO8 SPI0_CE0_N	ADC: CS (pin 1)
	Ground	25	26	GPIO7 SPI0_CE1_N	
Dual H-Bridge: in3	ID_SD I2C ID EEPROM	27	28	ID_SC I2C ID EEPROM	
Dual H-Bridge: in4	GPIO5	29	30	Ground	
Dual H-Bridge: enB	GPIO6	31	32	GPIO12	Tilt Motor (PWM)
Pan Motor (PWM)	GPIO13	33	34	Ground	
	GPIO19	35	36	GPIO16	
	GPIO26	37	38	GPIO20	
	Ground	39	40	GPIO21	

Software Setup: Package Install Commands



```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install python3-pip

python3 -m pip install --upgrade pillow

pip install matplotlib

pip3 install "numpy == 1.15.0" --user

pip3 install adafruit-circuitpython-mcp3xxx

sudo apt-get install rpi.gpio

sudo pip3 install adafruit-circuitpython-lsm303-accel

sudo pip3 install rpi_ws281x adafruit-circuitpython-neopixel
sudo python3 -m pip install --force-reinstall adafruit-blinka
```

Raspberry Pi Code (Python)

For easier viewing, see the GitHub page: https://github.com/Ctru14/Dia-Bot

DiaBotGUI.py – Top-level file creates GUI and begins other processes

```
import sys
import os
import tkinter as tk
from tkinter import *
from tkinter.scrolledtext import ScrolledText
from PIL import ImageTk, Image
import time
from datetime import datetime
import threading
import multiprocessing
import math
import enum
from random import *

import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk
from matplotlib.figure import Figure

# Dia-Bot specific imports
import DataCollection
import DataDisplay
import DataProcessing
import Alerts
from Positioning import Point3d
import Positioning
from Alerts import Alert
from Alerts import AlertDataType
from Alerts import AlertMetric
from Alerts import AlertRange
from Alerts import AlertTracker
from Alerts import AlertsTop
from Threads import DiaThread
from Threads import DiaProcess

piConnected = True
try:
    import PiInterface
```

```
except Exception as e:
    piConnected = False
    print(f"Error importing PiInterface: {e}")

# Debugging function - run a function and report how long it takes
def elapsedTime(func, *args):
    startTime = time.time_ns()
    func()
    elapsedTimeNs = time.time_ns() - startTime
    print("ElapsedTime (" + str(func.__name__) + ") = " + str(elapsedTimeNs / 1_000_000) + " ms")


class DiaBotGUI():

    def __init__(self, *args, **kwargs):
        # Initialize necessary variables
        self.cameraOn = False
        self.pid = os.getpid()
        self.top = tk.Tk()
        self.top.title('Dia-Bot')
        self.speed = IntVar()
        self.zoom = IntVar()

        # Threading control
        self.visualsRefreshTime = 2 # Number of seconds between visuals refresh
        self.programRunning = True
        self.startTime = time.time_ns()

        # Define GUI frames
        #   Primary sections
        self.controlFrame = tk.Frame(self.top, width=450, height=900)#, bg='orange')
        self.dataFrame = tk.Frame(self.top, width=1120, height=270)#, bg='blue')
        self.videoFrame = tk.Frame(self.top, width=1120, height=630)#, bg='red')
        #   Individual Control Frames
        self.movementControls = tk.Frame(self.controlFrame, width=450, height=280)#, bg='blue')
        self.cameraControls = tk.Frame(self.controlFrame, width=450, height=280)
        self.alertControls = tk.Frame(self.controlFrame, width=450, height=450)


        # Create queues
        self.processingQueue = multiprocessing.Queue()
        self.soundLevelAlertIOQueue = multiprocessing.Queue()
        self.vibrationAlertIOQueue = multiprocessing.Queue()
        self.tempAlertIOQueue = multiprocessing.Queue()
        self.alertIOqueues = [self.soundLevelAlertIOQueue, self.vibrationAlertIOQueue,
self.tempAlertIOQueue]


        # Data collection (Must be created in constructor to guaranteee use in Alerts)
        self.soundLevelSamplingRate = 100
        self.soundLevelFields, self.soundLevelDataQueue, self.soundLevelVisualQueue,
self.soundLevelCollection = DiaBotGUI.createDataFields(
            DataCollection.SoundLevelCollection, "Sound Level", "dB", self.soundLevelSamplingRate,
self.startTime)

        self.vibrationSamplingRate = 100
        self.vibrationFields, self.vibrationDataQueue, self.vibrationVisualQueue,
self.vibrationCollection = DiaBotGUI.createDataFields(
            DataCollection.VibrationCollection, "Vibration", "m/s2", self.vibrationSamplingRate,
self.startTime)

        self.temperatureSamplingRate = 1/5
        self.temperatureFields, self.temperatureDataQueue, self.temperatureVisualQueue,
self.temperatureCollection = DiaBotGUI.createDataFields(
            DataCollection.TemperatureCollection, "Temperature", "°C",
self.temperatureSamplingRate, self.startTime)

        # Position is handled differently! Still creates fields, but no extra processes
        self.positionSamplingRate = 10
        self.positionFields, self.positionDataQueue, self.positionVisualQueue, self.positionCollection
= DiaBotGUI.createDataFields(
            DataCollection.PositionCollection, "Position", "m", self.positionSamplingRate,
self.startTime)

        # Group of all the data classes
        self.dataFieldsClassList = [self.soundLevelFields, self.vibrationFields, self.positionFields,
self.temperatureFields]



    # Closes relevant processes and stops GPIO
    def exit(self):
        self.camera.close()
        self.top.destroy
        quit()


    # -------------------------- GUI SETUP CODE --------------------------
    def setupGuiFrames(self):
        self.top.resizable(width=False, height=False)
        self.top.geometry("1920x1016")

        # Build the frames
        self.setupDataPane()
        self.setupControlsPane()
        self.setupVideoPane()

        # Place frames
        self.bindEvents()
        self.placeFrames()



    # ------------------ Controls Pane ----------------------

    # --- Callback functions ---
    def speedChanged(self, event):
        PiInterface.speed=self.speed.get()


    # --- Controls pane setup function ---
    def setupControlsPane(self):
        # Controls top text
        self.controlsLabel = tk.Label(self.controlFrame, text="Controls", font="none 18 bold")#,
bg="orange")
        self.controlsLabel.grid(row=1, column=1, columnspan=8)
        self.controlsLabel.config(anchor=CENTER)
        self.controlFrame.grid_rowconfigure(1, minsize=60)

        # Get images for menu icons
        self.importMenuImages()

        # Create individual controls panes
        self.setupMovementControls()
        self.setupCameraControls()
        self.setupAlertControls()



    # ----- Movement controls -----
    def setupMovementControls(self):
        self.movementControls.grid(row=2, column=1, rowspan=2, columnspan=10)
        Label(self.movementControls, text="Movement", anchor=CENTER, font="none 14
bold").grid(row=1, column=1, columnspan=9)

        # Speed slider
        tk.Label(self.movementControls, text="Speed", anchor=CENTER, font="bold").grid(row=2,
column=2)
        self.speedScale = tk.Scale(self.movementControls, from_=100, to=0, orient=tk.VERTICAL,
variable=self.speed, command=self.speedChanged, length=150, showvalue=1, sliderlength=20)
        self.speedScale.grid(row=3, column=2, rowspan=4)
        self.speedScale.set(50)

        # Directional buttons
        tk.Label(self.movementControls, text="Direction", anchor=CENTER,
font="bold").grid(row=2, column=4, columnspan=3)
        self.setupMovementDirectionalButtons()

        # Stop and lock buttons
        tk.Label(self.movementControls, text="Mode", anchor=CENTER, font="bold").grid(row=2,
column=9)
        tk.Button(self.movementControls, text="Stop", command=PiInterface.stopMovement,
anchor=CENTER, fg="red", font="16").grid(row=3, column=9)
        tk.Button(self.movementControls, text="Lock", command=PiInterface.lock, anchor=CENTER,
font="16").grid(row=5, column=9)

        self.movementControls.grid_columnconfigure(1, minsize=10)
        for i in range(2,10):
            self.movementControls.grid_columnconfigure(i, minsize=20)


    def importMenuImages(self):
        # Directional arrows
        self.arrowUp = ImageTk.PhotoImage(Image.open("Assets/Arrow-Up.jpg").resize((30, 30)))
        self.arrowUpW = ImageTk.PhotoImage(Image.open("Assets/Arrow-Up-W.jpg").resize((30,
30)))
        self.arrowUpLeft = ImageTk.PhotoImage(Image.open("Assets/Arrow-Up-
Left.jpg").resize((30, 30)))
        self.arrowUpRight = ImageTk.PhotoImage(Image.open("Assets/Arrow-Up-
Right.jpg").resize((30, 30)))
        self.arrowDown = ImageTk.PhotoImage(Image.open("Assets/Arrow-Down.jpg").resize((30,
30)))
        self.arrowDownS = ImageTk.PhotoImage(Image.open("Assets/Arrow-Down-
S.jpg").resize((30, 30)))
        self.arrowDownLeft = ImageTk.PhotoImage(Image.open("Assets/Arrow-Down-
Left.jpg").resize((30, 30)))
```

```
        self.arrowDownRight = ImageTk.PhotoImage(Image.open("Assets/Arrow-Down-Right.jpg").resize((30, 30)))
        self.arrowLeft = ImageTk.PhotoImage(Image.open("Assets/Arrow-Left.jpg").resize((30, 30)))
        self.arrowLeftA = ImageTk.PhotoImage(Image.open("Assets/Arrow-Left-A.jpg").resize((30, 30)))
        self.arrowRight = ImageTk.PhotoImage(Image.open("Assets/Arrow-Right.jpg").resize((30, 30)))
        self.arrowRightD = ImageTk.PhotoImage(Image.open("Assets/Arrow-Right-D.jpg").resize((30, 30)))
        # Other
        self.cameraIcon = ImageTk.PhotoImage(Image.open("Assets/Camera-Icon.jpg").resize((30, 30)))
        self.deleteIcon = ImageTk.PhotoImage(Image.open("Assets/Delete-Icon.jpg").resize((22, 22)))
        self.clearIcon = ImageTk.PhotoImage(Image.open("Assets/Clear-Icon.jpg").resize((22, 22)))



    # Directional buttons
    def setupMovementDirectionalButtons(self):

        # Forward
        self.moveForwardButton = tk.Button(self.movementControls, image=self.arrowUpW, anchor=CENTER, font="16")
        self.moveForwardButton.bind("<ButtonPress>", PiInterface.moveForwardPress)
        self.moveForwardButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveForwardButton.grid(row=3, column=5)

        # Forward-Left
        self.moveForwardLeftButton = tk.Button(self.movementControls, image=self.arrowUpLeft, anchor=CENTER, font="16")
        self.moveForwardLeftButton.bind("<ButtonPress>", PiInterface.moveForwardLeftPress)
        self.moveForwardLeftButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveForwardLeftButton.grid(row=3, column=4)

        # Forward-Right
        self.moveForwardRightButton = tk.Button(self.movementControls, image=self.arrowUpRight, anchor=CENTER, font="16")
        self.moveForwardRightButton.bind("<ButtonPress>", PiInterface.moveForwardRightPress)
        self.moveForwardRightButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveForwardRightButton.grid(row=3, column=6)

        # Backward
        self.moveBackwardButton = tk.Button(self.movementControls, image=self.arrowDownS,
anchor=CENTER, font="16")
        self.moveBackwardButton.bind("<ButtonPress>", PiInterface.moveBackwardPress)
        self.moveBackwardButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveBackwardButton.grid(row=5, column=5)

        # Backward-Left
        self.moveBackwardLeftButton = tk.Button(self.movementControls,
image=self.arrowDownLeft, anchor=CENTER, font="16")
        self.moveBackwardLeftButton.bind("<ButtonPress>", PiInterface.moveBackwardLeftPress)
        self.moveBackwardLeftButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveBackwardLeftButton.grid(row=5, column=4)

        # Backward-Right
        self.moveBackwardRightButton = tk.Button(self.movementControls,
image=self.arrowDownRight, anchor=CENTER, font="16")
        self.moveBackwardRightButton.bind("<ButtonPress>",
PiInterface.moveBackwardRightPress)
        self.moveBackwardRightButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveBackwardRightButton.grid(row=5, column=6)

        # Left
        self.moveLeftButton = tk.Button(self.movementControls, image=self.arrowLeftA,
anchor=CENTER, font="16")
        self.moveLeftButton.bind("<ButtonPress>", PiInterface.moveLeftPress)
        self.moveLeftButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveLeftButton.grid(row=4, column=4)

        # Right
        self.moveRightButton = tk.Button(self.movementControls, image=self.arrowRightD,
anchor=CENTER, font="16")
        self.moveRightButton.bind("<ButtonPress>", PiInterface.moveRightPress)
        self.moveRightButton.bind("<ButtonRelease>", PiInterface.moveRelease)
        self.moveRightButton.grid(row=4, column=6)

        # Keyboard Buttons
        self.top.bind("<KeyPress-w>", PiInterface.moveForwardPress)
        self.top.bind("<KeyRelease-w>", PiInterface.moveRelease)
        self.top.bind("<KeyPress-s>", PiInterface.moveBackwardPress)
        self.top.bind("<KeyRelease-s>", PiInterface.moveRelease)
        self.top.bind("<KeyPress-a>", PiInterface.moveLeftPress)
        self.top.bind("<KeyRelease-a>", PiInterface.moveRelease)
        self.top.bind("<KeyPress-d>", PiInterface.moveRightPress)
        self.top.bind("<KeyRelease-d>", PiInterface.moveRelease)
```

```
    # ----- Camera Controls -----
    def setupCameraControls(self):
        self.cameraControls.grid(row=5, column=1, rowspan=1, columnspan=10)
        tk.Label(self.cameraControls, text="Camera", anchor=CENTER, font="none 14
bold").grid(row=1, column=1, columnspan=9)

        # Directional buttons
        tk.Label(self.cameraControls, text="Angle", anchor=CENTER, font="bold").grid(row=2,
column=2, columnspan=3)
        tk.Button(self.cameraControls, image=self.arrowUp, command=PiInterface.cameraUp,
anchor=CENTER, font="16").grid(row=3, column=3)
        tk.Button(self.cameraControls, image=self.arrowDown, command=PiInterface.cameraDown,
anchor=CENTER, font="16").grid(row=5, column=3)
        tk.Button(self.cameraControls, image=self.arrowLeft, command=PiInterface.cameraLeft,
anchor=CENTER, font="16").grid(row=4, column=2)
        tk.Button(self.cameraControls, image=self.arrowRight, command=PiInterface.cameraRight,
anchor=CENTER, font="16").grid(row=4, column=4)
        tk.Button(self.cameraControls, image=self.cameraIcon, command=self.takePhoto,
anchor=CENTER, font="16").grid(row=4, column=3)

        # Stop and lock buttons
        tk.Label(self.cameraControls, text="Light", anchor=CENTER, font="bold").grid(row=2,
column=6)
        tk.Button(self.cameraControls, text="On", command=PiInterface.ledOn, anchor=CENTER,
font="16").grid(row=4, column=6)
        tk.Button(self.cameraControls, text="Off", command=PiInterface.ledOff, anchor=CENTER,
font="16").grid(row=5, column=6)

        # Zoom slider
        tk.Label(self.cameraControls, text="Zoom", anchor=CENTER, font="bold").grid(row=2,
column=8)
        self.zoomScale = tk.Scale(self.cameraControls, from_=100, to=0, orient=tk.VERTICAL,
variable=self.zoom, length=150, showvalue=0, sliderlength=20)
        self.zoomScale.grid(row=3, column=8, rowspan=4)
        self.zoomScale.set(50)

        self.cameraControls.grid_columnconfigure(1, minsize=10)
        for i in range(2,10):
            self.cameraControls.grid_columnconfigure(i, minsize=20)

    # TK button function to capture and save image
    def takePhoto(self, *args):
        dtFormat = "{:%Y%m%d-%H%M%S}"
        timeString = dtFormat.format(datetime.now())
        fileName = f"img-{timeString}.jpg"
        path = os.path.join(self.photosPath, fileName)
        PiInterface.captureImage(path)

    # ----- Alert Controls -----
    def setupAlertControls(self):
        self.alertControls.grid(row=7, column=1, rowspan=1, columnspan=10)
        tk.Label(self.alertControls, text="Alert Trackers", anchor=CENTER, font="none 14
bold").grid(row=1, column=1, columnspan=9)

        # Extra TK frame to display just the alert trackers
        self.alertTrackersFrame = tk.Frame(self.alertControls, width=400)

        # Create each alert tracker instance and add frames to the UI
        self.alertsTop = AlertsTop(self.alertControls, self.alertTrackersFrame, self.processingQueue,
self.alertIOqueues, self.deleteIcon, self.clearIcon, self.alertsText, PiInterface.captureImage)

        self.vibrationAlertTracker = AlertTracker(self.alertsTop, self.alertTrackersFrame,   "Vibration",
AlertDataType.Vibration,   AlertRange.Above,   AlertMetric.Average, self.vibrationAlertIOQueue,
self.deleteIcon, self.clearIcon)
        self.temperatureAlertTracker = AlertTracker(self.alertsTop, self.alertTrackersFrame,
"Temperature", AlertDataType.Temperature, AlertRange.Between, AlertMetric.Average,
self.tempAlertIOQueue, self.deleteIcon, self.clearIcon)

        self.alertsTop.addTracker(self.vibrationAlertTracker)
        self.alertsTop.addTracker(self.temperatureAlertTracker)

        self.alertTrackersFrame.grid(row=2, column=1, columnspan=12)

        # Press this button to confirm and lock in Alert changes
        self.confirmButton = tk.Button(self.alertControls, text="Confirm Trackers",
command=self.alertsTop.updateAlerts)
        self.confirmButton.grid(row=3, column=8, columnspan=2)

        # Add frame to add new trackers
        self.newAlertsFrame = self.alertsTop.buildNewTrackerFrame(self.alertControls)
        self.newAlertsFrame.grid(row=4, column=1, columnspan=11)

        self.alertControls.grid_columnconfigure(1, minsize=10)
        for i in range(2,10):
            self.alertControls.grid_columnconfigure(i, minsize=20)
        for i in range(2, 5):
            self.alertControls.grid_rowconfigure(i, minsize=30)
    # ------------------ Data Pane -----------------------

    # Main method to setup data pane with each data category
    def setupDataPane(self):
        tk.Label(self.dataFrame, text="Data", font="none 18 bold").grid(row=1, column=1,
columnspan=50)

        # Individual Frames
        self.soundLevelFrame = tk.Frame(self.dataFrame, width=350, height=350)
        self.vibrationFrame = tk.Frame(self.dataFrame, width=350, height=350)
        self.temperatureFrame = tk.Frame(self.dataFrame, width=350, height=350)
        self.positionFrame = tk.Frame(self.dataFrame, width=350, height=350)
        self.alertsDisplayFrame = tk.Frame(self.dataFrame, width=350, height=350)
        self.dataFrames = [self.soundLevelFrame, self.vibrationFrame, self.temperatureFrame,
self.positionFrame]

        # Sound Level
        self.soundLevelDisplayClass = DataDisplay.DataDisplay(self.soundLevelFields,
self.soundLevelFrame, self.soundLevelVisualQueue)
        self.soundLevelDisplayClass.tkAddDataPane()
        self.soundLevelFrame.grid(row=2, column=1, padx=10)

        # Vibration
        self.vibrationDisplayClass = DataDisplay.DataDisplay(self.vibrationFields,
self.vibrationFrame, self.vibrationVisualQueue)
        self.vibrationDisplayClass.tkAddDataPane()
        self.vibrationFrame.grid(row=2, column=2, padx=10)

        # Temperature
        self.tempDisplayClass = DataDisplay.TemperatureDisplay(self.temperatureFields,
self.temperatureFrame, self.temperatureVisualQueue)
        self.tempDisplayClass.tkAddDataPane()
        self.temperatureFrame.grid(row=2, column=3, padx=10)

        # Position
        self.zeroPositionQueue = multiprocessing.Queue()
        self.positionDisplayClass = DataDisplay.PositionDisplay(self.positionFields,
self.positionFrame, self.positionVisualQueue, self.zeroPositionQueue)
        self.positionDisplayClass.tkAddDataPane()
        self.positionFrame.grid(row=2, column=4, padx=10)
        # Alerts scrolled text
        self.alertsDisplayLabel = tk.Label(self.alertsDisplayFrame, text="Alerts", font="none 12
bold")
        self.alertsDisplayLabel.pack()
        self.alertsText = ScrolledText(self.alertsDisplayFrame, width=40, height=9, font = "none 14")
        self.alertsText.pack()
        self.alertsDisplayFrame.grid(row=2, column=5, padx=10)


    def createDataFields(CollectionType, name, units, samplingRate, startTime):
        dataQueue = multiprocessing.Queue()
        visualQueue = multiprocessing.Queue()
        collection = CollectionType(name, units, samplingRate, startTime, dataQueue)
        fields = DataCollection.DataFields(name, units, samplingRate, startTime,
collection.alertDataType)
        return (fields, dataQueue, visualQueue, collection)



    # ------------------ Video Pane -----------------------

    # --- Callback functions ---

    def setupVideoPane(self):
        self.testImg = ImageTk.PhotoImage(Image.open("Assets/Video-Frame.jpg").resize((1380,
715)))
        self.imgLabel = Label(self.videoFrame, image=self.testImg)
        self.imgLabel.grid(row=1, column=1)

    # -------------- Put it all together ---------------
    def placeFrames(self):
        # Place the frames
        self.controlFrame.place(relx=0.01, rely=0.01, anchor=tk.NW)
        self.dataFrame.place(relx=0.3, rely=0.01, anchor=tk.NW)
        self.videoFrame.place(x=450, y=300, anchor=tk.NW)


    # ---------- Threading functions ----------

    def bindEvents(self):
        self.top.bind("<<visualsEvent>>", self.updateVisualsWrapper)
        self.top.bind("<<alertsEvent>>", self.updateAlertsHandler)
    # --- Update Visuals Handlers ---

    # Sends update visuals event to TK
    def generateEvent(self, eventString, *args):
        if self.programRunning:
            try:
                self.top.event_generate(eventString)
            except Exception as e:
                print(f"Unable to update visuals! Error in event_generate: {e}")

    # Wrapper function around the handler for updating the visuals
    def updateVisualsWrapper(self, event):
        #elapsedTime(self.updateVisualsHandler)
        self.updateVisualsHandler()

    # Any data visual which requires manual update (new graphs use animations to update
automatically)
    def updateVisualsHandler(self):
        # Only temperature view needs updating
        self.tempDisplayClass.updateVisual()
        self.positionDisplayClass.updateVisual()

    # --- Update Alerts Handlers ---
    def updateAlertsHandler(self, event):
        try:
            self.alertsTop.distributeProcessedData((self.positionDisplayClass.curX,
self.positionDisplayClass.curY, self.positionDisplayClass.curZ))
        except Exception as e:
            print(f"Exception thrown in update alerts: {e}")

    # Calibrate accelerometer for the following purposes:
    #  1. Rotate data so gravity is in the -Y direction
    #  2. Find the average magnitude of gravity then re-scale to ~9.8 m/s2
    def calibrateAccelerometer(self):
        accelerometer = PiInterface.Accelerometer()
        testPoints = []
        # Collect 3s of data for calibration
        t0 = time.time()
        while time.time() - t0 < 3:
            data = accelerometer.readAccData()
            point = Point3d(time.time(), data[0], data[1], data[2])
            testPoints.append(point)
            time.sleep(.01)
        # Return rotation angles and magnitude of gravity
        angX, angZ, gravMag = Positioning.calibrateAcc(testPoints)
        return (angX, angZ, gravMag)



    # ----- Main method for GUI - Starts extra threads and processes and other programs ----
    def startProgram(self):
        # Calibrate accelerometer
        try:
            print("Calibrating accelerometer, keep Dia-Bot still...")
            self.accCalibration = self.calibrateAccelerometer()
            print(f"...calibration complete: {self.accCalibration}")
        except Exception as e:
            print(f"Error calibrating accelerometer: {e}")
            self.accCalibration = (0, 0, 1)

        # Create GUI
        self.setupGuiFrames()

        # Create and add processes and threads
        useProcesses = True
        shutdownRespQueue = multiprocessing.Queue()

        # ----- Create other processes and threads -----
        # GUI updating threads
        visualThread = DiaThread("visualThread", False, self.startTime, shutdownRespQueue,
1/self.visualsRefreshTime, self.generateEvent, "<<visualsEvent>>")
        alertThread = DiaThread("alertThread", False, self.startTime, shutdownRespQueue, 1/2,
self.generateEvent, "<<alertsEvent>>")

        # Data collection threads (separate processes)
        self.adcCollection = DataCollection.ADCCollection("ADC Collection",
self.soundLevelSamplingRate, self.soundLevelDataQueue, self.temperatureDataQueue)
        adcCollectionProcess = DiaThread("adcCollectionProcess", useProcesses, self.startTime,
shutdownRespQueue, self.soundLevelSamplingRate, self.adcCollection.readAndSendData)
        vibrationCollectionProcess = DiaThread("vibrationCollectionProcess", useProcesses,
self.startTime, shutdownRespQueue,  self.vibrationSamplingRate,
self.vibrationCollection.readAndSendData)

        # Sound and Temperature are merged into ADC collection - leaving this here in case this
ever changes
        #soundCollectionProcess = DiaThread("soundCollectionProcess", useProcesses,
self.startTime, shutdownRespQueue, self.soundLevelSamplingRate,
self.soundLevelCollection.readAndSendData)
```

```
        #temperatureCollectionProcess = DiaThread("temperatureCollectionProcess", useProcesses,
self.startTime, shutdownRespQueue, self.temperatureSamplingRate,
self.temperatureCollection.readAndSendData)
        #threads = [visualThread, alertThread, soundCollectionProcess, vibrationCollectionProcess,
temperatureCollectionProcess]

        threads = [visualThread, alertThread, adcCollectionProcess, vibrationCollectionProcess]

        # Parent processes for data processing
        soundLevelShutdownInitQueue = multiprocessing.Queue()
        soundLevelProcess = DiaProcess(self.soundLevelFields, soundLevelShutdownInitQueue,
shutdownRespQueue, DataProcessing.SoundLevelProcessing,
                                        False, self.soundLevelDataQueue, self.soundLevelVisualQueue,
self.processingQueue, self.soundLevelAlertIOQueue)

        vibrationShutdownInitQueue = multiprocessing.Queue()
        vibrationProcess = DiaProcess(self.vibrationFields, vibrationShutdownInitQueue,
shutdownRespQueue, DataProcessing.VibrationProcessing,
                                        False, self.vibrationDataQueue, self.vibrationVisualQueue,
self.processingQueue, self.vibrationAlertIOQueue, self.positionVisualQueue,
self.zeroPositionQueue, self.accCalibration)

        tempShutdownInitQueue = multiprocessing.Queue()
        temperatureProcess = DiaProcess(self.temperatureFields, tempShutdownInitQueue,
shutdownRespQueue, DataProcessing.TemperatureProcessing,
                                        False, self.temperatureDataQueue, self.temperatureVisualQueue,
self.processingQueue, self.tempAlertIOQueue)

        parentProcesses = [soundLevelProcess, vibrationProcess, temperatureProcess]


        for process in parentProcesses:
            process.startProcess()

        for t in threads:
            t.startThread()
        self.programRunning = True # Used in updateVisuals()

        # Start camera preview
        try:
            PiInterface.start_camera()
            self.cameraOn = True
        except Exception as e:
            print(f"Error starting camera: {e}")
```

```
    self.cameraOn = False


# Add folder for photos
self.rootPath = os.path.dirname(__file__)
self.photosPath = os.path.join(self.rootPath, "Photos")
if not os.path.exists(self.photosPath):
    print(f"Photos path does not exist - creating: {self.photosPath}")
    os.mkdir(self.photosPath)

# ----- Blocking call: Begin TK mainloop -----
print("-------- BEGINING TK MAINLOOP --------")
self.top.mainloop()
self.programRunning = False
print("-------- TK MAINLOOP ENDED: ENDING WORKER THREADS --------")

# After UI closed: cleanup!

# Send signals to end all threads and processes
# Shutdown extra processes properly
for process in parentProcesses: # DiaProcess
    process.beginShutdown()

threadRunningCount = 0
for t in threads:
    threadRunningCount += 1
    t.endThread()

try:
    PiInterface.stopGpio()
except Exception as e:
    print(f"Error stopping GPIO: {e}")

if self.cameraOn:
    PiInterface.stop_camera()


# Try to join all processes after completion
print(f"Joining parent processes...") # DiaProcess
for process in parentProcesses:
    print(f"Joining parent process {process.name} (alive = {process.is_alive()})...")
    process.joinProcess(1)
    print(f"...parent process {process.name} attempted join. (alive = {process.is_alive()})")
```

```
        # Collect signals for ending threads and join
        DiaThread.waitForThreadsEnd(threads, shutdownRespQueue, "Main", self.pid, 20)

        print(f"All threads ended in {self.pid}:Parent process! Joining...")
        DiaThread.joinAllThreads(threads)

        print("Thank you for using Dia-Bot")


def main():
    gui = DiaBotGUI()
    gui.startProgram()

if __name__ == "__main__":
    main()
```

DataDisplay.py – Classes that create and show visuals for each data type

```
import sys
import os
import tkinter as tk
from tkinter import *
from PIL import ImageTk, Image
import time
import threading
import multiprocessing
import math
import enum
from random import *

import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk
from matplotlib.figure import Figure

import matplotlib.animation as animation
from collections import deque

import matplotlib.dates as mdates

import DataProcessing


import matplotlib.ticker as ticker



# Credit for this solution for millisecond time display goes to StackOverflow user "hemmelig"
# https://stackoverflow.com/questions/11107748/showing-milliseconds-in-matplotlib
class PrecisionDateFormatter(ticker.Formatter):
    """
    Extend the `matplotlib.ticker.Formatter` class to allow for millisecond
    precision when formatting a tick (in days since the epoch) with a
    `~datetime.datetime.strftime` format string.

    """


    def __init__(self, fmt, precision=2, tz=None):
        """
        Parameters
        ----------
        fmt : str
            `~datetime.datetime.strftime` format string.
        """
        from matplotlib.dates import num2date
        if tz is None:
            from matplotlib.dates import _get_rc_timezone
            tz = _get_rc_timezone()
        self.num2date = num2date
        self.fmt = fmt
        self.tz = tz
        self.precision = precision

    def __call__(self, x, pos=0):
        #if x == 0:
        #    raise ValueError("DateFormatter found a value of x=0, which is "
        #                     "an illegal date; this usually occurs because "
        #                     "you have not informed the axis that it is "
        #                     "plotting dates, e.g., with ax.xaxis_date()")
        #
        dt = self.num2date(x, self.tz)
        ms = dt.strftime("%f")[:self.precision]

        return dt.strftime(self.fmt).format(ms=ms)

    def set_tzinfo(self, tz):
        self.tz = tz



# Owned by main TK process to display the data
class DataDisplay:

    def __init__(self, fields, tkTop, visualQueue):
        self.name = fields.name
        self.units = fields.units
        self.tkTop = tkTop
        self.visualQueue = visualQueue
        # Finf max length of data to display: 3 seconds worth or 250, whichever is smaller
        self.displayDataLen = int((fields.samplingRate * 3)/10)*10
        self.displayDataLen = min(self.displayDataLen, 250)
        self.t = deque([], maxlen=self.displayDataLen)
        self.data = deque([], maxlen=self.displayDataLen)
    # Create and add the Tkinter pane for data visualization - may be overwritten for those without graphs
    def tkAddDataPane(self, *args):
        # Top label
        tk.Label(self.tkTop, text=self.name, font="none 12 bold").grid(row=1, column=1, columnspan=5)
        # Initialize the plot
        self.fig = Figure(figsize=(3,2.5), dpi=80)
        self.fig.patch.set_facecolor("#DBDBDB")
        self.plot1 = self.fig.add_subplot(111)
        self.plot1.set_ylabel(self.units)
        self.line, = self.plot1.plot([], [], lw=2)
        self.plot1.xaxis.set_major_locator(matplotlib.ticker.MaxNLocator(2))
        self.plot1.xaxis.set_major_formatter(PrecisionDateFormatter("%H:%M:%S.{ms}"))
        self.canvas = FigureCanvasTkAgg(self.fig, master=self.tkTop)
        self.canvas.draw()
        self.canvas.get_tk_widget().grid(row=2, column=1, rowspan=3, columnspan=4)
        self.ani = animation.FuncAnimation(
            self.fig,
            self.appendNewData,
            interval=2000, # Time (ms) between graph updates
            repeat=True)
        self.ani._start()


    def appendNewData(self, *args):
        while not self.visualQueue.empty():
            t, data = self.visualQueue.get()
            self.t.append(t)
            self.data.append(data)
        if len(self.t) > 0:
            self.line.set_data(self.t, self.data)
            self.plot1.set_ylim(min(self.data), max(self.data))
            self.plot1.set_xlim(self.t[0], self.t[-1])
        return self.line,


# Displaying X/Y/Z data for positioning
class PositionDisplay(DataDisplay):

    def __init__(self, fields, tkTop, visualQueue, zeroPositionQueue = 0):
        super().__init__(fields, tkTop, visualQueue)
        self.posMutex = threading.Lock()
        self.curX = 0.0
        self.curY = 0.0
        self.curZ = 0.0
        self.displayX = StringVar()
        self.displayY = StringVar()
        self.displayZ = StringVar()
        self.updateVisual()
        self.zeroPositionQueue = zeroPositionQueue

    def readNewData(self):
        pos = (self.curX, self.curY, self.curZ)
        while not self.visualQueue.empty():
            t, pos = self.visualQueue.get()
        self.posMutex.acquire()
        self.curX, self.curY, self.curZ = pos
        self.posMutex.release()

    # Called periodically by UI thread
    def updateVisual(self):
        self.readNewData()
        self.posMutex.acquire()
        self.displayX.set("X: {:.2f} m".format(self.curX))
        self.displayY.set("Y: {:.2f} m".format(self.curY))
        self.displayZ.set("Z: {:.2f} m".format(self.curZ))
        self.posMutex.release()

    def zeroPosition(self, *args):
        self.zeroPositionQueue.put("ZERO")

    # Overwrite data visuals method: no graph needed
    def tkAddDataPane(self):
        # Top label
        self.topLabel = tk.Label(self.tkTop, text=self.name, font="none 12 bold")
        self.topLabel.grid(row=1, column=1, columnspan=5)
        # Add temperature text and display button
        self.xLabel = tk.Label(self.tkTop, textvariable=self.displayX, font="none 14")
        self.yLabel = tk.Label(self.tkTop, textvariable=self.displayY, font="none 14")
        self.zLabel = tk.Label(self.tkTop, textvariable=self.displayZ, font="none 14")
        self.xLabel.grid(row=3, column=1, columnspan=5)
        self.yLabel.grid(row=4, column=1, columnspan=5)
        self.zLabel.grid(row=5, column=1, columnspan=5)
        # Button to reset position to zero
        self.zeroPosButton = tk.Button(self.tkTop, text = "Zero Position", command =
self.zeroPosition)
        self.zeroPosButton.grid(row=6, column=1, columnspan=5)
```

```
# Displaying text and button for Temperature data
class TemperatureDisplay(DataDisplay):

    def __init__(self, fields, tkTop, visualQueue):
        super().__init__(fields, tkTop, visualQueue)
        self.viewFarenheit = False
        self.currentTempCelsius = 0
        self.currentTempFarenheit = 0
        self.tempDisplayText = StringVar()
        self.tempDisplayText.set(self.getDisplayText())
        self.tempViewButtonText = StringVar()
        self.tempViewButtonText.set("View Farenheit")

    def getDisplayText(self):
        if self.viewFarenheit:
            tempF = "{:.1f}".format(self.currentTempFarenheit)
            return f"{tempF} °F"
        else:
            tempC = "{:.1f}".format(self.currentTempCelsius)
            return f"{tempC} °C"

    def switchTempView(self):
        print(f"Switching temp view! Temp = {self.tempDisplayText.get()}, Button =
{self.tempViewButtonText.get()}")
        if self.viewFarenheit:
            # Currently Farenheit --> Switch to Celsius
            self.viewFarenheit = False
            self.tempDisplayText.set(self.getDisplayText())
            self.tempViewButtonText.set("View Farenheit")
        else:
            # Currently Celsius --> Switch to Farenheit
            self.viewFarenheit = True
            self.tempDisplayText.set(self.getDisplayText())
            self.tempViewButtonText.set("View Celsius")

    # UI thread - collect new temperature data
    def readNewData(self):
        while not self.visualQueue.empty():
            t, dataC = self.visualQueue.get()
            self.currentTempCelsius = dataC
            self.currentTempFarenheit = dataC * 9 / 5 + 32
    # Called by UI thread to update temperature printout
    def updateVisual(self):
        self.readNewData()
        self.tempDisplayText.set(self.getDisplayText())

    # Overwrite data visuals method: no graph needed
    def tkAddDataPane(self):
        # Top label
        self.topLabel = tk.Label(self.tkTop, text=self.name, font="none 12 bold")
        self.topLabel.grid(row=1, column=1, columnspan=5)
        # Add temperature text and display button
        self.tempLabel = tk.Label(self.tkTop, textvariable=self.tempDisplayText, font="none 14")
        self.tempLabel.grid(row=3, column=1, columnspan=5)
        self.switchTempViewButton = tk.Button(self.tkTop, textvariable=self.tempViewButtonText,
command=self.switchTempView)
        self.switchTempViewButton.grid(row=4, column=1, columnspan=5)
```

DataCollection.py – Classes that collect data from Pi Interface functions and accumulate for processing

```
import sys
import time
from datetime import datetime
import threading
import multiprocessing
import math
from random import *

from PiInterface import Accelerometer
from PiInterface import ADC

from Alerts import AlertDataType

class DataFields:

    def __init__(self, name, units, samplingRate, startTime, alertDataType):
        self.name = name
        self.units = units
        self.samplingRate = samplingRate
        self.samplingTime = 1/samplingRate
        self.startTime = startTime
        self.alertDataType = alertDataType



# Class is used in both the GPIO collection process and the processing process for queue collection
class DataCollection(DataFields):
    def __init__(self, name, units, samplingRate, startTime, dataQueue, alertDataType):
        super().__init__(name, units, samplingRate, startTime, alertDataType)
        #self.dataMutex = threading.Lock()
        self.startTime = startTime
        self.t = []
        self.data = []
        self.dataQueue = dataQueue

    # Used in processing process - appends new data point to the data array
    def addData(self, t, data):
        #self.dataMutex.acquire()
        self.t.append(t)
        self.data.append(data)
        #self.dataMutex.release()

    # Retrieves all new data from the queue and appends it to the array - called by processing
process
    def getAndAddData(self, *args):
        while not self.dataQueue.empty():
            t, data = self.dataQueue.get()
            self.addData(t, data)

    # Reads data from given function - called by data collection process
    def readAndSendData(self, *args):
        t = datetime.now()
        data = self.readData()
        self.dataQueue.put((t, data))
```

```
# DEPRECATED - SOUND LEVEL USES ADC COLLECTION CLASS
class SoundLevelCollection(DataCollection):

    def __init__(self, name, units, samplingRate, startTime, dataQueue):
        super().__init__(name, units, samplingRate, startTime, dataQueue,
AlertDataType.SoundLevel)
        #self.adc = ADC()

    def readData(self):
        num = uniform(-10, 10)
        return num

    #def readData(self):
        #return self.adc.readSoundData()




class VibrationCollection(DataCollection):

    def __init__(self, name, units, samplingRate, startTime, dataQueue):
        super().__init__(name, units, samplingRate, startTime, dataQueue, AlertDataType.Vibration)
        self.accelerometer = Accelerometer()

    # addData override in DataProcessing class!

    def readData(self):
        return self.accelerometer.readAccData()
```

```
# DEPRECATED - POSITION IS NO LONGER HANDLED LIKE THE OTHER DATA TYPES! (updated
by vibration)
#  Leaving this here in case a new position method is found in the future
class PositionCollection(DataCollection):

    def __init__(self, name, units, samplingRate, startTime, dataQueue):
        return super().__init__(name, units, samplingRate, startTime, dataQueue,
AlertDataType.Position)

    def readData(self):
        pos = (uniform(-10, 10), uniform(-10, 10), uniform(-10, 10))
        #print("Reading position! - " + str(pos))
        return pos



class TemperatureCollection(DataCollection):

    def __init__(self, name, units, samplingRate, startTime, dataQueue):
        super().__init__(name, units, samplingRate, startTime, dataQueue,
AlertDataType.Temperature)
        #self.adc = ADC()

    def readData(self):
        return self.adc.readTemperatureData()

    # Reads data from given function (DEPRECATED - INSTEAD USES ADC COLLECTION)
    def readAndSendData(self, *args):
        print("!!!!! UNEXPECTED USE OF TemperatureCollection readAndSendData FUNCTION CALL !!!!!")
        t = datetime.now()
        data = self.readData()
        self.dataQueue.put((t, data))
        #self.visualQueue.put((t, data))


class ADCCollection():

    def __init__(self, name, soundLevelSamplingRate, soundLevelDataQueue, temperatureDataQueue):
        self.samplingRate = soundLevelSamplingRate
        self.soundLevelDataQueue = soundLevelDataQueue
        self.temperatureDataQueue = temperatureDataQueue
        self.tempLoopNum = 0
        self.tempLoopMax = self.samplingRate * 4 # Approximately 4s/Temp
        self.adc = ADC()


    def readAndSendData(self, *args):
        t = datetime.now()
        soundData = self.adc.readSoundData()
        self.soundLevelDataQueue.put((t, soundData))
        if self.tempLoopNum == 0:
            tempData = self.adc.readTemperatureData()
            self.temperatureDataQueue.put((t, tempData))
        self.tempLoopNum += 1
```

```
if self.tempLoopNum == self.tempLoopMax:
    self.tempLoopNum = 0
```

DataProcessing.py – Calculate various processing metrics

```
import sys
import os
import csv
import tkinter as tk
from tkinter import *
from PIL import ImageTk, Image
import time
import threading
import multiprocessing
import math
import numpy as np
from random import *

import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg, NavigationToolbar2Tk
from matplotlib.figure import Figure

from DataCollection import DataCollection
from DataCollection import VibrationCollection
from Alerts import AlertDataType
from Alerts import AlertMetric
from Positioning import Point3d
import Positioning
import Threads
```

```
class DataProcessing(DataCollection):

    def __init__(self, alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue):
        super().__init__(name, units, samplingRate, startTime, dataQueue, alertDataType)
        self.alertDataType = alertDataType
        self.visualQueue = visualQueue
        self.processingQueue = processingQueue
        self.dataMutex = threading.Lock()
        self.lastIdx = 0



    # ----- Data Processing functions -----
    def average(self, idxLo, idxHi):
        avg = np.mean(self.data[idxLo:idxHi])
        #print(f"Calculating average between {idxLo} and {idxHi}: {avg}")
        return avg

    def maximum(self, idxLo, idxHi):
        max = np.max(self.data[idxLo:idxHi])
        #print(f"Calculating maximum between {idxLo} and {idxHi}: {max}")
        return max

    def minimum(self, idxLo, idxHi):
        min = np.min(self.data[idxLo:idxHi])
        #print(f"Finding minimum between {idxLo} and {idxHi}: {min}")
        return min
```

```
def frequency(self, idxLo, idxHi):
    self.fft = np.fft.fft(self.data[idxLo:idxHi])
    self.freqs = np.fft.fftfreq(len(self.fft))
    self.idx = np.argmax(np.abs(self.fft))
    freq = self.freqs[self.idx]
    #print(f"Finding {self.name} frequency between {idxLo} and {idxHi}: {freq}")
    return freq

def magnitude(self, idxLo, idxHi):
    mag = np.abs(self.fft[self.idx])
    #print(f"Finding {self.name} magnitude between {idxLo} and {idxHi}: {mag}")
    return mag

def addDataToVisualQueue(self, idxHi):
    while self.lastIdx <= idxHi:
        self.visualQueue.put((self.t[self.lastIdx], self.data[self.lastIdx]))
        self.lastIdx += 1



def mainProcessing(self, *args):
    # Calculate all processing values and put them into the queue
    idxHi = len(self.t)-1
    if idxHi > 0:
        t = self.t[idxHi]
        idxLo = max(0, int(idxHi - (10 * self.samplingRate)))
        avg = self.average(idxLo, idxHi)
        maximum = self.maximum(idxLo, idxHi)
        minimum = self.minimum(idxLo, idxHi)
```

```
            freq = self.frequency(idxLo, idxHi)
            mag = self.magnitude(idxLo, idxHi)
            self.processingQueue.put((self.alertDataType, avg, maximum, minimum, freq, mag, t, (idxLo, idxHi)))
            self.addDataToVisualQueue(idxHi)
        return idxHi


class SoundLevelProcessing(DataProcessing):


    def __init__(self, alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue):
        return super().__init__(alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue)




class VibrationProcessing(DataProcessing):


    def __init__(self, alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue, positionQueue, zeroPositionQueue, accCalibration):
        super().__init__(alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue)
        self.positionQueue = positionQueue
        self.dataRaw = [] # Point3d
        self.lastPosIdx = 0
        self.curVel = Point3d(0, 0, 0, 0)
        self.curPos = Point3d(0, 0, 0, 0)
        self.angX, self.angZ, self.gravMag = accCalibration
        self.zeroPositionQueue = zeroPositionQueue
```

```
# DataCollection method! Overridden  instead due to inheritance complications
# Used in processing process - appends new data point to the data array
def addData(self, t, data):
    #self.dataMutex.acquire()
    self.t.append(t)
    newPoint = Point3d(t.timestamp(), data[0], data[1], data[2])
    newPoint = newPoint.rotX(self.angX)
    newPoint = newPoint.rotZ(self.angZ)
    newPoint.y = newPoint.y + self.gravMag
    newPoint.multiply(9.80665/self.gravMag)
    self.dataRaw.append(newPoint)
    self.data.append(newPoint.mag())
    #self.dataMutex.release()

# Calculate all processing values and put them into the queue
# Processing done on magnitude!
def mainProcessing(self, *args):
    idxHi = super().mainProcessing()
    # Integrate the new vibration acceleration data
    if idxHi > 0:
        self.calculatePosition(idxHi)

def calculatePosition(self, idxHi):
    if not self.zeroPositionQueue.empty():
        msg = self.zeroPositionQueue.get()
        if msg == "ZERO":
            self.curVel.multiply(0.0)
            self.curPos.multiply(0.0)
        # Track position up to the last index
        if self.lastPosIdx == 0 and len(self.dataRaw) > 0:
            self.curVel.t = self.dataRaw[0].t
            self.curPos.t = self.dataRaw[0].t
        #print(f"Track position from idx {self.lastPosIdx} up to idx {idxHi}:")
        while self.lastPosIdx < idxHi:
            acc = self.dataRaw[self.lastPosIdx]
            #print(f"  idx {self.lastPosIdx}: Pos={self.curPos}, Vel={self.curVel}, Acc={acc}")
            Positioning.writeNextIntegralPoint(self.curVel, acc.t, acc.x, acc.y, acc.z)
            Positioning.writeNextIntegralPoint(self.curPos, self.curVel.t, self.curVel.x, self.curVel.y, self.curVel.z)
            self.lastPosIdx += 1
        # Write new position to the queue
        self.positionQueue.put((self.curPos.t, (self.curPos.x, self.curPos.y, self.curPos.z)))




class PositionProcessing(DataProcessing):


    def __init__(self, alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue):
        return super().__init__(alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue)


    def mainProcessing(self, *args):
        # Calculate all processing values and put them into the queue
        idxHi = len(self.t)-1
        if idxHi > 0:
            t = self.t[idxHi]
            idxLo = max(0, int(idxHi - (10 * self.samplingRate)))
            self.addDataToVisualQueue(idxHi)



class TemperatureProcessing(DataProcessing):


    def __init__(self, alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue):

        super().__init__(alertDataType, name, units, samplingRate, startTime, isPlotted, dataQueue, visualQueue, processingQueue)
```

Alerts.py – Receives alert metrics and manages their display

```
import sys
import os
import tkinter as tk
from tkinter import *
import time
import threading
import multiprocessing
import uuid
from math import *
from random import *
import enum
from copy import deepcopy

# Dia-Bot classes
import DataCollection
from Positioning import Point3d

# Types of alerts - range, processing metric, data type

# Starts from 0 to index into AlertsTop lists
class AlertDataType(enum.IntEnum):
   SoundLevel = 0
   Vibration = 1
   Temperature = 2
   Position = 3

# Starts from 1 to index into ProcessingQueue tuple (which has the data type as the first member)
```

```
class AlertMetric(enum.IntEnum):
    Average = 1
    Maximum = 2
    Minimum = 3
    Frequency = 4
    Magnitude = 5

class AlertRange(enum.IntEnum):
    Above = 0
    Between = 1
    Below = 2



class Alert:

    def __init__(self, alertDataType, alertTime, alertRange, alertMetric, tripValue, indices, trackerName = ""):
        self.id = str(uuid.uuid4())
        self.alertDataType = alertDataType
        self.time = alertTime
        self.alertRange = alertRange
        self.alertMetric = alertMetric
        self.tripValue = tripValue
        self.indices = indices
        self.trackerName = trackerName



alertDataTypes = (AlertDataType.SoundLevel.name, AlertDataType.Vibration.name, AlertDataType.Temperature.name)
```

```
fullDataTypes = (AlertDataType.SoundLevel.name, AlertDataType.Vibration.name,
AlertDataType.Temperature.name, AlertDataType.Position.name)
alertMetrics = (AlertMetric.Average.name, AlertMetric.Maximum.name,
AlertMetric.Minimum.name, AlertMetric.Frequency.name, AlertMetric.Magnitude.name)
alertRanges = (AlertRange.Above.name, AlertRange.Between.name, AlertRange.Below.name)
dataTypeUnits = ("dB", "m/s2", "°C", "m")


class AlertTracker:


    def __init__(self, alertsTop, alertControlsFrame, name, alertDataType, alertRange, alertMetric,
alertIOqueue, deleteIcon, clearIcon, width=400, height=100):
        # Initialize data variables
        self.name = name
        self.alertsTop = alertsTop
        self.alertEnabled = BooleanVar()
        self.alertDataType = alertDataType
        self.thresholdUnits = dataTypeUnits[int(alertDataType)]
        self.alertRange = alertRange
        self.alertMetric = alertMetric
        self.alertRangeName = StringVar()
        self.alertRangeName.set(self.alertRange.name)
        self.alertIOqueue = alertIOqueue
        self.alerts = []
        self.deleteIcon = deleteIcon
        self.clearIcon = clearIcon
        self.errorActive = False
        self.alertsMutex = threading.Lock()
        self.alertsDataPath = self.alertsTop.alertsDataPaths[int(self.alertDataType)]
        self.dateTimeFormat = "{:%Y%m%d-%H%M%S}"
        # Threshold levels
        self.belowValue = nan
        self.aboveValue = nan
        self.betweenLoValue = nan
        self.betweenHiValue = nan

        # Strings to hold the alert thresholds
        self.thresholdString1 = StringVar()
        self.thresholdString2 = StringVar()

        # Create TKinter frame
        self.frame = tk.Frame(alertControlsFrame, width=width, height=height)
        self.nameEnableButton = tk.Checkbutton(self.frame, text=self.name, variable=self.alertEnabled, anchor="w", justify=LEFT, font="none 11")
        self.dataTypeLabel = tk.Label(self.frame, text=self.alertDataType.name, anchor="w", justify=LEFT, font="none 11")
        self.metricLabel = tk.Label(self.frame, text=self.alertMetric.name, anchor="w", justify=LEFT, font="none 11")
        self.notificationLabel = tk.Label(self.frame, text="None", anchor=CENTER, font="none 11", fg="black")
        self.rangeMenu = tk.OptionMenu(self.frame, self.alertRangeName, *alertRanges, command=self.alertRangeChanged)
        self.input1 = tk.Entry(self.frame, justify=CENTER, width=6, font="none 11", textvariable=self.thresholdString1)
        self.input2 = tk.Entry(self.frame, justify=CENTER, width=6, font="none 11", textvariable=self.thresholdString2)
        self.unitsLabel = tk.Label(self.frame, text=self.thresholdUnits, anchor="w", justify=RIGHT, font="none 11")
        self.clearButton = tk.Button(self.frame, image=self.clearIcon, command=self.clearAlerts)
        self.deleteButton = tk.Button(self.frame, image=self.deleteIcon, command=self.deleteTracker)
```

```
# Builds and returns the alert frame in self.frame
def getAlertFrame(self):
    #print(f"Creating and returning alert row for {self.name})
    self.nameEnableButton.place(x=0, y=0, anchor=tk.NW)
    self.dataTypeLabel.place(x=135, y=0, anchor=tk.NW)
    self.metricLabel.place(x=260, y=0, anchor=tk.NW)
    self.notificationLabel.place(x=400, y=0, anchor=tk.NE)

    # Alert ranges
    self.rangeMenu.place(x=0, y=30, anchor=tk.NW)

    # Input entry fields: Only show the second entry for 'Between' mode
    self.input1.place(x=110, y=30, anchor=tk.NW)
    if (self.alertRange == AlertRange.Between):
        self.input2.place(x=185, y=30, anchor=tk.NW)

    # Units
    self.unitsLabel.place(x=260, y=30, anchor=tk.NW)

    # Clear and Delete buttons
    self.clearButton.place(x=365, y=30, anchor=tk.NE)
    self.deleteButton.place(x=400, y=30, anchor=tk.NE)

    # Alert notification
    return self.frame

def deleteTracker(self):
    print(f"Delete tracker: {self.name}")
```

```
        self.frame.destroy()
        self.alertsTop.removeTracker(self)

    # Callback function for changing the alert type
    def alertRangeChanged(self, typeName):
        self.alertRangeName.set(typeName)
        self.alertRange = AlertRange[typeName]
        if typeName == AlertRange.Above.name:
            print(f"Alert type changed to {self.alertRange} ({typeName}): change above limit!")
            self.input2.place_forget()
        elif typeName == AlertRange.Below.name:
            print(f"Alert type changed to {self.alertRange} ({typeName}): change below limit!")
            self.input2.place_forget()
        elif typeName == AlertRange.Between.name:
            print(f"Alert type changed to {self.alertRange} ({typeName}): change between limits and add the entry box")
            self.input2.place(x=185, y=30, anchor=tk.NW)



    def confirmUpdates(self):
        try:
           threshold1 = float(self.thresholdString1.get())
           if self.alertRange == AlertRange.Above:
               self.aboveValue = threshold1
           elif self.alertRange == AlertRange.Below:
               self.belowValue = threshold1
        except:
            print(f"Error: cannot convert string {self.thresholdString1.get()} to a number")
        if self.alertRange == AlertRange.Between:
            try:
               threshold2 = float(self.thresholdString2.get())
               thresholdLo = min(threshold1, threshold2)
               thresholdHi = max(threshold1, threshold2)
               self.thresholdString1.set(str(thresholdLo))
               self.thresholdString2.set(str(thresholdHi))
               self.betweenLoValue = thresholdLo
               self.betweenHiValue = thresholdHi
            except:
                print(f"Error: cannot convert string {self.thresholdString2.get()} to a number")

    def clearAlerts(self):
        self.errorActive = False
        self.alertsMutex.acquire()
        self.alerts.clear()
        self.alertsMutex.release()
        self.notificationLabel.place_forget()
        self.notificationLabel = tk.Label(self.frame, text="None", anchor=CENTER, font="none 11", fg="black")
        self.notificationLabel.place(x=400, y=0, anchor=tk.NE)


    def setErrorLabel(self):
        #timeString = time.strftime("%a, %d %b %Y %H:%M:%S", time.localtime(alert.time)) # Add %Z to show time zone
        self.notificationLabel.place_forget()
        self.notificationLabel = tk.Label(self.frame, text=f"Error({len(self.alerts)})", anchor=CENTER, font="none 11 bold", fg="red")
        self.notificationLabel.place(x=400, y=0, anchor=tk.NE)


    def getAlertsDisplayText(self, alert):
        alertTime = "{:%H:%M:%S}".format(alert.time)
        value = "{:.{}f}".format(alert.tripValue, 5)
        return f"{self.name} #{len(self.alerts)} - {alertTime} \n - {self.alertDataType.name}
{self.alertMetric.name} = {value} ({self.alertRange.name} {self.tripValue})\n"

    def addNewAlertText(self, text):
        self.alertsTop.alertsTextDisplay.insert(INSERT, text)

    def checkAlertCondition(self, value):
        if self.alertRange == AlertRange.Above:
            if value > self.aboveValue:
                self.tripValue = self.aboveValue
                return True
        elif self.alertRange == AlertRange.Below:
            if value < self.belowValue:
                self.tripValue = self.belowValue
                return True
        elif self.alertRange == AlertRange.Between:
            if value < self.betweenHiValue and value > self.betweenLoValue:
                self.tripValue = (self.betweenLoValue, self.betweenHiValue)
                return True
        return False

    def checkForAlerts(self, t, value, indices, position):
        if self.alertEnabled.get():
            errorFound = False
            # Checks tracker thresholds to compare this value
            if self.checkAlertCondition(value):
                errorFound = True
        #isNewAlert = False
        if not self.errorActive:
            self.errorActive = True
            isNewAlert = True
            newAlert = Alert(self.alertDataType, t, self.alertRange, self.alertMetric, value, indices, self.name)
            self.alertsMutex.acquire()
            self.alerts.append(newAlert)
            self.alertsMutex.release()
            print(f"Alert #{len(self.alerts)} found in {self.name} tracker at time {t}! {self.alertMetric.name}={value} {self.alertRange.name} {self.tripValue}")
            self.addNewAlertText(self.getAlertsDisplayText(newAlert))
            self.setErrorLabel()
        else:
            isNewAlert = False
            print(f"Error {self.alerts[-1].id} currently active in {self.name}! Append data to file...")
            self.alertsMutex.acquire()
            newAlert = deepcopy(self.alerts[-1])
            self.alertsMutex.release()
            newAlert.indices = indices
        # Alert found: Make new directory if it doesn't already exist
        # Construct directory name:  YYYYMMDD-hhmmss_Metric_Range_ID/
        trackerName = self.name.replace(" ", "")
        #timeString = time.strftime(self.timeFormat, time.gmtime(alert.time)) # time.gmtime(alert.time).strftime(self.timeFormat)
        timeString = self.dateTimeFormat.format(newAlert.time)#.strftime(self.timeFormat, time.gmtime(alert.time)) # time.gmtime(alert.time).strftime(self.timeFormat)
        alertDirName = f"{trackerName}_{timeString}_{self.alertMetric.name}_{self.alertRange.name}"
        alertDirPath = os.path.join(self.alertsDataPath, alertDirName)
```

```
                if not os.path.exists(alertDirPath):
                    # Create new alert directory
                    os.mkdir(alertDirPath)
                self.alertIOqueue.put((newAlert, position, alertDirPath, isNewAlert))
                self.alertsTop.takeImageFunction(os.path.join(alertDirPath, f"img-
{self.dateTimeFormat.format(t)}.jpg"))
            # Check if active error needs to be reset
            if self.errorActive and not errorFound:
                print(f"Previously active error in {self.name} was not tripped - resetting active")
                self.errorActive = False



# Top-level class to contain AlertTrackers
# Receives processing info from queue and sends it to each relevant tracker
class AlertsTop:

    def __init__(self, alertControlsFrame, alertTrackersFrame, processingQueue, alertIOqueues,
deleteIcon, clearIcon, alertsTextDisplay, takeImageFunction):
        self.alertControlsFrame = alertControlsFrame
        self.alertTrackersFrame = alertTrackersFrame
        self.processingQueue = processingQueue
        self.alertIOqueues = alertIOqueues
        self.deleteIcon = deleteIcon
        self.clearIcon = clearIcon
        self.alertsTextDisplay = alertsTextDisplay
        self.position = (0.0, 0.0, 0.0) # Sent with Alerts in IO queue
        self.takeImageFunction = takeImageFunction
        # Sort trackers in lists based on their data type
        self.soundLevelTrackers = []
        self.vibrationTrackers = []
        #self.positionTrackers = []
        self.temperatureTrackers = []
        self.trackers = [self.soundLevelTrackers, self.vibrationTrackers, self.temperatureTrackers]
        #self.trackers = [self.soundLevelTrackers, self.vibrationTrackers, self.positionTrackers, self.temperatureTrackers]
        # TK variables for the Add New Alert frame
        self.nameEntryVar = StringVar()
        self.nameEntryVar.set("Tracker Name")
        self.newDataTypeVar = StringVar()
        self.newMetricVar = StringVar()
        # Create new directory for alerts, if it does not exist yet
        self.rootPath = os.path.dirname(__file__)
        self.alertsPath = os.path.join(self.rootPath, "Alerts")
        if not os.path.exists(self.alertsPath):
            print(f"Alerts path does not exist - creating: {self.alertsPath}")
            os.mkdir(self.alertsPath)
        self.alertsDataPaths = []
        for name in alertDataTypes:
            alertsDataPath = os.path.join(self.alertsPath, name)
            self.alertsDataPaths.append(alertsDataPath)
            if not os.path.exists(alertsDataPath):
                print(f"{name} alerts data path does not exist - creating: {alertsDataPath}")
                os.mkdir(alertsDataPath)


    def addTracker(self, tracker):
        # Add tracker to a list based on the data type
        self.trackers[tracker.alertDataType].append(tracker)
        tracker.getAlertFrame().pack()


    # Delete tracker from UI and Top
    def removeTracker(self, tracker):
        trackersList = self.trackers[tracker.alertDataType]
        found = False
        for i in range(len(trackersList)):
            if trackersList[i] == tracker:
                found = True
                trackersList.pop(i)
                break
        if not found:
            print(f"Error in alertsTop.removeTracker: ({tracker.name}) not found!")


    def buildNewTrackerFrame(self, alertControlsFrame, width=400, height=100):
        # Create TKinter frame
        self.newTrackerFrame = tk.Frame(alertControlsFrame, width=width, height=height)
        self.newTrackerLabel = tk.Label(self.newTrackerFrame, text="Add New Alert:", anchor=CENTER, font="none 11", fg="black")
        self.newTrackerLabel.grid(row=1, column=1, columnspan=4)
        self.nameEntry = tk.Entry(self.newTrackerFrame, justify=CENTER, width=15, font="none 11", textvariable=self.nameEntryVar)
        self.nameEntry.grid(row=1, column=5, columnspan=4)
        self.dataTypeMenu = tk.OptionMenu(self.newTrackerFrame, self.newDataTypeVar, *alertDataTypes, command=self.alertDataTypeChanged)
        self.dataTypeMenu.grid(row=2, column=3, columnspan=3)
        self.metricMenu = tk.OptionMenu(self.newTrackerFrame, self.newMetricVar, *alertMetrics, command=self.alertMetricChanged)
        self.metricMenu.grid(row=2, column=6, columnspan=3)
        self.addButton = tk.Button(self.newTrackerFrame, text="+",
command=self.buildAndAddTracker)
        self.addButton.grid(row=1, column=10, rowspan=2, columnspan=2)
        return self.newTrackerFrame

    # Callback function for selecting a new alert data type
    def alertDataTypeChanged(self, typeName):
        self.newDataTypeVar.set(typeName)
        self.newDataType = AlertDataType[typeName]

    # Callback function for selecting a new alert metric
    def alertMetricChanged(self, metricName):
        self.newMetricVar.set(metricName)
        self.newMetric = AlertMetric[metricName]

    # Callback button for "+" new tracker - take UI input to build and add a new tracker
    def buildAndAddTracker(self):
        if len(self.newDataTypeVar.get()) > 1 and len(self.newMetricVar.get()) > 1:
            newTracker = AlertTracker(self, self.alertTrackersFrame, self.nameEntryVar.get(),
self.newDataType, AlertRange.Above, self.newMetric, self.alertIOqueues[int(self.newDataType)],
self.deleteIcon, self.clearIcon)
            self.addTracker(newTracker) # Add to existing list
            # Clear new tracker frame of the previous name name
            self.nameEntryVar.set("Tracker Name")
            return newTracker
        else:
            print("Error in buildAndAddTracker: data type or metric not selected!")

    # Accept UI changes to existing alert trackers to change tracker behavior
    def updateAlerts(self):
```

```
    for trackerList in self.trackers:
        for tracker in trackerList:
            tracker.confirmUpdates()

# Check processing queue for new metrics and distribute to proper trackers
def distributeProcessedData(self, position):
    # Check processing queue for new data
    self.position = position
    while not self.processingQueue.empty():
        processed = self.processingQueue.get()
        dataType = processed[0]
        alertTime = processed[6]
        indices = processed[7]
        for tracker in self.trackers[int(dataType)]:
            value = processed[int(tracker.alertMetric)]
            tracker.checkForAlerts(alertTime, value, indices, position)
```

FileIO.py – Writes relevant alert data to the file system upon receiving alert context

```
import sys
import os
import csv
from PIL import ImageTk, Image
import time
import threading
import math
import enum

# Dia-Bot specific imports
import DataCollection
import DataDisplay
import DataProcessing
import Alerts
import Threads
from Alerts import Alert
from Alerts import AlertDataType
from Alerts import AlertMetric
from Alerts import AlertRange
from Alerts import AlertTracker
from Alerts import AlertsTop



class FileIO:

    # New FileIO class runs in the same process context as the rest of the data
```

```
def __init__(self, fields, alertIOqueue, processing):
    self.name = fields.name
    self.units = fields.units
    self.samplingRate = fields.samplingRate
    self.alertDataType = fields.alertDataType
    self.processing = processing
    self.alertIOqueue = alertIOqueue
    self.timeFormat = "%y%m%d-%H%M%S"
    self.dateTimeFormat = "{:%Y%m%d-%H%M%S}"


def writeAlertData(self, alert, position, alertDirPath, isNewAlert):
    idxLo = alert.indices[0]
    idxHi = alert.indices[1]
    csvDataPath = os.path.join(alertDirPath, f"raw_data.csv")
    csvPositionPath = os.path.join(alertDirPath, f"position.csv")
    if not os.path.exists(alertDirPath):
        print(f"ERROR IN writeAlertData: dir {alertDirPath} does not exist!")
    # New alert or update?
    if not isNewAlert:
        # Alert already exists - update image and data
        print(f"Alert path exists! Updating raw data in {alertDirPath}")
        with open(csvDataPath, 'a', newline='') as csvDataFile:
            writer = csv.writer(csvDataFile)
            for i in range(idxLo, idxHi):
                writer.writerow([self.processing.t[i], self.processing.data[i]])
        with open(csvPositionPath, 'a', newline='') as csvPositionFile:
            writer = csv.writer(csvPositionFile)
            writer.writerow([self.processing.t[i], position[0], position[1], position[2]])
    else:
        # New alert - create new directory and write data
        print(f"Writitng new {self.name} alert data idxs=({idxLo}..{idxHi}) to {alertDirPath}")
        # Create and write raw data to CSV
        with open(csvDataPath, 'w', newline='') as csvDataFile:
            writer = csv.writer(csvDataFile)
            writer.writerow(["Time", f"{self.name} Data"])
            for i in range(idxLo, idxHi):
                writer.writerow([self.processing.t[i], self.processing.data[i]])
        with open(csvPositionPath, 'a', newline='') as csvPositionFile:
            writer = csv.writer(csvPositionFile)
            writer.writerow(["Time", "Position-X", "Position-Y", "Position-Z"])
            writer.writerow([self.processing.t[i], position[0], position[1], position[2]])


def alertIO(self, *args):
    #print(f"Alert IO starting - args = {args}")
    while not self.alertIOqueue.empty():
        alert, position, alertDirPath, isNewAlert = self.alertIOqueue.get()
        self.writeAlertData(alert, position, alertDirPath, isNewAlert)
        print(f"Writing {self.alertDataType.name} to file - Alert IO in {self.name}! {alert}")
```

Positioning.py – Calibrate, filter, and process accelerometer data for position tracking

```
import numpy as np
import math



class Point3d:

    def __init__(self, time, x, y, z):
        self.t = time
        self.x = x
        self.y = y
        self.z = z

    def mag(self):
        return math.sqrt(self.x**2 + self.y**2 + self.z**2)

    def __add__(self, other):
        return Point3d(max(self.t, other.t), self.x+other.x, self.y+other.y, self.z+other.z)

    def __div__(self, other):
        return Point3d(max(self.t, other.t), self.x/other.x, self.y/other.y, self.z/other.z)

    def normalize(self):
        mag = self.mag()
        return Point3d(self.t, self.x/mag, self.y/mag, self.z/mag)

    def rotX(self, ang):
        c = math.cos(ang)
    s = math.sin(ang)
    m = np.array([
        [1, 0, 0,    0],
        [0, c, -1*s, 0],
        [0, s, c,    0],
        [0, 0, 0,    1]])
    v = np.array([self.x,self.y,self.z,1])
    newV = np.matmul(m,v)
    return Point3d(self.t, newV[0], newV[1], newV[2])

def rotZ(self, ang):
    c = math.cos(ang)
    s = math.sin(ang)
    m = np.array([
        [c, -1*s, 0, 0],
        [s, c,    0, 0],
        [0, 0,    1, 0],
        [0, 0,    0, 1]])
    v = np.array([self.x,self.y,self.z,1])
    newV = np.matmul(m,v)
    return Point3d(self.t, newV[0], newV[1], newV[2])

def multiply(self, num):
    self.x = self.x * num
    self.y = self.y * num
    self.z = self.z * num

def __repr__(self):
        return "[t: %f, x: %f, y: %f, z: %f]" % (self.t, self.x, self.y, self.z)

# Given the current integral point and the next value to interate, return the new pooint
def getNextIntegralPoint(prev, t, int_x, int_y, int_z):
    x = prev.x + (t-prev.t)*int_x
    y = prev.y + (t-prev.t)*int_y
    z = prev.z + (t-prev.t)*int_z
    # Point3d: {t, x, y, z}
    return Point3d(t, x, y, z)

# In-place version of getNextIntegralPoint - prev is updated
def writeNextIntegralPoint(prev, t, int_x, int_y, int_z):
    x = prev.x + (t-prev.t)*int_x
    y = prev.y + (t-prev.t)*int_y
    z = prev.z + (t-prev.t)*int_z
    # Point3d: {t, x, y, z}
    prev.t = t
    prev.x = x
    prev.y = y
    prev.z = z

# Add a singular next datapoint to an integral
def addIntegralDatapoint(points, t, int_x, int_y, int_z):
    newPoint = getNextIntegralPoint(points[-1], t, int_x, int_y, int_z)
    points.append(new_pos)
    return

# Take an integral over an entire list of points
```

```
def integrate(points, t0=-1, cx=0, cy=0, cz=0, idxLo=0):
    # Initial point: time of initial raw point, '+ C' data provided in parameters defaults to zero
    i=idxLo
    # If given an initial time, use it. If not, use the first point's time
    t = points[i].t
    if t0 != -1:
        t = t0
    integral = [Point3d(t, cx, cy, cz)]
    while i<len(points)-1:
        addIntegralDatapoint(integral, points[i+1].t, points[i].x, points[i].y, points[i].z)
        i = i+1
    return integral

# Calibrate acceleration data to be able to rotate and remove gravity vector
# Returns filtering metrics: rotation angles in X and Z directions + gravity magnitude
def calibrateAcc(accRaw):

    # Find the first index of movement
    idx = 0
    grav = Point3d(0, 0, 0, 0)
    mags = []
    while idx < len(accRaw):
        grav = grav + accRaw[idx]
        mags.append(accRaw[idx].mag())
        idx = idx + 1
    # Find the average magnitude direction of gravity
    gravMag = np.mean(mags)
    grav.t = 0
```

```
 grav = grav.normalize()
 print("Average: " + str(grav))

 # Rotate data so gravity is in the -Y direction
  # First rotate about the X axis
 angX = math.acos(-1*grav.y/Point3d(0,0,grav.y,grav.z).mag())
 gravX = grav.rotX(angX)
 print("\nRotation around the X axis! Expect Z=0")
 print(f"Angle: {angX}  in degrees: {angX*180/math.pi}")
 print(f"New Point: {gravX}")

# Then about the Z axis
 angZ = math.acos(-1*gravX.y) #/gravX.mag())
 gravZ = gravX.rotZ(-1*angZ)
 print("\nRotation around the Z axis! Expect (0, -1, 0)")
 print(f"Angle: {angZ}  in degrees: {angZ*180/math.pi}")
 print(f"New Point: {gravZ}")

 return angX, angZ, gravMag
```

Threads.py – Handles multiprocessing and multithreading applications for the rest of the program

```
import sys

import os

import time

import math

import threading

import multiprocessing

import FileIO



class DiaThread():


    def __init__(self, name, useProcess, startTime, shutdownRespQueue, freqHz, loopFunction,
*args):

        self.startTime = startTime

        self.threadRunning = False

        self.threadEnded = False

        self.shutdownInitQueue = multiprocessing.Queue() # Process receives ending message on
this queue

        self.shutdownRespQueue = shutdownRespQueue # Thread/process sends message to
parent when completed

        self.loopFreq = freqHz

        self.loopTIme = 1/freqHz

        self.name = name

        self.useProcess = useProcess

        print(f"Create and add new loop thread: {loopFunction.__name__}")

        if useProcess:

            self.thread = multiprocessing.Process(name=self.name, target=self.loopAtFrequency,
args=(freqHz, self.shutdownInitQueue, loopFunction, args))

            self.thread.daemon = True
        else:
            self.thread = threading.Thread(name=self.name, target=self.loopAtFrequency,
args=(freqHz, self.shutdownInitQueue, loopFunction, args))
            self.thread.daemon = True



    # Wrapper to other functions which loops
    def loopAtFrequency(self, freqHz, shutdownInitQueue, loopFunction, *args):
        print(f"Starting thread {self.name} with args {args} (len {len(args)}) at {freqHz} Hz -
{self.thread}")
        loopTime = 1/freqHz
        pid = os.getpid()
        while self.threadRunning:
            loopStartTime = time.time()
            #print(f"Calling loopFunction {loopFunction.__name__}: {args}")
            loopFunction(*args)
            loopEndTime = time.time()
            loopTimeTaken = loopEndTime - loopStartTime
            timeRemaining = loopTime - (loopTimeTaken)
            if timeRemaining > 0:
                time.sleep(timeRemaining)
            else:
                print(f"Thread {self.name} took longer to execute ({loopTimeTaken} s) than its given
time({loopTime} s)! Assigning {loopTime}s sleep")
                time.sleep(loopTime)
            # For processes, check the shutdown queue for a stop message
            # (threads keep self.threadRunning in the same context, so queues are unnecessary)
            if self.useProcess:
                if not shutdownInitQueue.empty():
                    msg = shutdownInitQueue.get()
                    #print(f"shutdownInitQueue msg: {msg}")
                    if msg == "END_THREAD":
                        self.threadRunning = False
        self.threadEnded = True
        self.shutdownRespQueue.put(("THREAD_ENDED", self.name))
        print(f"Loop ended! {self.name} ({pid})")

    def startThread(self):
        self.threadRunning = True
        self.thread.start()

    # Sets thread ending flag, but NON-BLOCKING
    def endThread(self):
        print(f"Ending thread! {self.name}")
        self.threadRunning = False
        if self.useProcess:
            self.shutdownInitQueue.put("END_THREAD")

    def join(self, *args):
        return self.thread.join(*args)

    def is_alive(self):
        return self.thread.is_alive()

    def terminate(self):
        if self.useProcess:
            return self.thread.terminate()
        else:
            print(f"Error - Only processes can use terminate() - {self.name} uses threading")


    # BLOCKING call to ensure all threads end
    def waitForThreadsEnd(threads, shutdownRespQueue, name, pid, maxLoops = float('inf')):
        threadRunningCount = len(threads)
        loops = 0
        while threadRunningCount > 0 and loops < maxLoops:
            # Check for thread ending messages every second
            loops += 1
            if not shutdownRespQueue.empty():
                msg, name = shutdownRespQueue.get()
                if msg == "THREAD_ENDED":
                    threadRunningCount -= 1
                    print(f"Shutdown message received within {pid}:{name} - waiting on {threadRunningCount} more!")
                else:
                    print(f"UNEXPECTED MESSAGE IN SHUTDOWN RESPONSE QUEUE: {msg}")
            else:
                time.sleep(1)
                if loops >= maxLoops:
                    print(f"Max time hit in waitForThreadsEnd! ({maxLoops} loops)")


    def joinAllThreads(threads):
        for t in threads:
            print(f"Joining DiaThread {t.name}...")
            t.join(1)
            if t.is_alive():
                print(f"DiaThread {t.name} did not join...terminating")
                t.terminate()
```

```
# Parent process starts a new process which spawns child threads
class DiaProcess():

    def __init__(self, fields, shutdownInitQueue, shutdownRespQueue, ProcessingType, isPlotted, dataQueue, visualQueue, processingQueue, alertIOqueue, positionQueue=0, zeroPositionQueue=0, accCalibration=0):
        self.name = fields.name.replace(" ", "")
        self.externalShutdownInitQueue = shutdownInitQueue # External - Receive shutdown message from main process
        self.externalShutdownRespQueue = shutdownRespQueue # External - Confirm shutdown to main process
        self.process = multiprocessing.Process(target=DiaProcess.beginDataProcessing, args=(fields, ProcessingType, isPlotted, dataQueue, visualQueue, processingQueue, alertIOqueue, shutdownInitQueue, positionQueue, zeroPositionQueue, accCalibration))
        self.process.daemon = True

    # Called from main process
    def startProcess(self):
        self.process.start()

    # Called from main process
    def beginShutdown(self):
        #print(f"Sending shutdown message to procuess {self.name}")
        self.externalShutdownInitQueue.put("END_PROCESS")

    # Called from main process
    def joinProcess(self, *args):
        self.process.join(*args)
    # Called from main process
    def is_alive(self):
        return self.process.is_alive()

    # Called internally by process
    def waitForShutdownMessage(externalShutdownInitQueue, loopTime):
        endMessageReceived = False
        while not endMessageReceived:
            while externalShutdownInitQueue.empty():
                time.sleep(loopTime)
            msg = externalShutdownInitQueue.get()
            if msg == "END_PROCESS":
                endMessageReceived = True



    # -------- Function to initialize data processing processes --------
    #   ----- This will be run in the context of the new process! -----
    def beginDataProcessing(fields, ProcessingType, isPlotted, dataQueue, visualQueue,
processingQueue, alertIOqueue, externalShutdownInitQueue, positionQueue,
zeroPositionQueue, accCalibration):
        pid = os.getpid()
        threadRunningCount = 0
        internalShutdownRespQueue = multiprocessing.Queue()
        name = fields.name.replace(" ", "")

        # Initialize DataProcessing class in new process context
        if name == "Vibration": # Send position queue to vibration processing
            processing = ProcessingType(fields.alertDataType, name, fields.units, fields.samplingRate, fields.startTime, isPlotted, dataQueue, visualQueue, processingQueue, positionQueue, zeroPositionQueue, accCalibration)
        else:
            processing = ProcessingType(fields.alertDataType, name, fields.units, fields.samplingRate, fields.startTime, isPlotted, dataQueue, visualQueue, processingQueue)


        fileIO = FileIO.FileIO(fields, alertIOqueue, processing)


        # Add child threads for data collection, visuals, and processing
        collectionThread = DiaThread(f"{name}CollectionThread", False, fields.startTime, internalShutdownRespQueue, fields.samplingRate, processing.getAndAddData)
        processingThread = DiaThread(f"{name}ProcessingThread", False, fields.startTime, internalShutdownRespQueue, .4, processing.mainProcessing)
        alertIOthread = DiaThread(f"{name}AlertIOThread", False, fields.startTime, internalShutdownRespQueue, .1, fileIO.alertIO)


        # Start worker threads
        threads = [collectionThread, processingThread, alertIOthread]#, visualThread]
        for t in threads:
            t.startThread()
            threadRunningCount += 1
            print(f"Starting thread {t.name} in {name}:{pid}")


        # LOOP HERE DURING EXECUTION - Wait for shutdown message - check every 3 seconds
        DiaProcess.waitForShutdownMessage(externalShutdownInitQueue, 3)


        # End threads - Send signal, NON-BLOCKING
        for t in threads:
            t.endThread()
```

```
# Collect Thread ending messages
DiaThread.waitForThreadsEnd(threads, internalShutdownRespQueue, name, pid)

# Threads ended - join me, and together, we will rule the galaxy...
print(f"All threads ended in {name}:{pid} - joining...")

DiaThread.joinAllThreads(threads)

print(f"DiaProcess {name}:{pid} completed.")
```

PiInterface.py – Contains wrapper functions called by other modules for GPIO control

```
import sys
import time
import threading
import math
from random import *
import picamera
import RPi.GPIO as GPIO
import pigpio

import board
import busio
import adafruit_lsm303_accel_edited as adafruit_lsm303_accel

import DCMotor
import DualHBridge
import DataCollection

import signal

import Positioning

import os
import digitalio
import adafruit_mcp3xxx.mcp3002 as MCP
from adafruit_mcp3xxx.analog_in import AnalogIn

import neopixel

from gpiozero import Servo
from gpiozero.pins.pigpio import PiGPIOFactory


# RPi GPIO Initializations

# GPIO Setup
gpioMode = GPIO.BCM
#gpioMode = GPIO.BOARD
GPIO.setwarnings(False)
GPIO.setmode(gpioMode)
pi = pigpio.pi()
camera = picamera.PiCamera()
cameraMutex = threading.Lock()
```

```
pixels = neopixel.NeoPixel(board.D18, 12)

# Motor Pins
motorIn1L = 24
motorIn2L = 23
motorEnL = 25
motorIn1R = 0
motorIn2R = 5
motorEnR = 6

GPIO.setup(motorIn1L, GPIO.OUT)
GPIO.setup(motorIn2L, GPIO.OUT)
GPIO.setup(motorEnL, GPIO.OUT)
GPIO.output(motorIn1L, GPIO.LOW)
GPIO.output(motorIn2L, GPIO.LOW)
pwmEnL=GPIO.PWM(motorEnL, 1000)
GPIO.setup(motorIn1R, GPIO.OUT)
GPIO.setup(motorIn2R, GPIO.OUT)
GPIO.setup(motorEnR, GPIO.OUT)
GPIO.output(motorIn1R, GPIO.LOW)
GPIO.output(motorIn2R, GPIO.LOW)
pwmEnR=GPIO.PWM(motorEnR, 1000)

pwmEnL.start(25)
pwmEnR.start(25)


# Camera Control
class CameraAngle:

    def __init__(self, tiltPin=12, panPin=13):
        self.factory = PiGPIOFactory()
        self.tilt = Servo(tiltPin, pin_factory=self.factory)
        self.pan = Servo(panPin, min_pulse_width=0.83/1000, max_pulse_width=1.55/1000,
pin_factory=self.factory)
        self.pan.mid()
        self.tilt.mid()
        self.tilt_value = 0;
        self.pan_value = 0;

    def tiltIncrement(self, num):
        if self.tilt_value < 0.95 and self.tilt_value > -0.95:
            self.tilt_value = self.tilt_value + num
            self.tilt.value = self.tilt_value
        elif self.tilt_value > 0.95 and num < 0:
            self.tilt_value = self.tilt_value + num
            self.tilt.value = self.tilt_value
        elif self.tilt_value < -0.95 and num > 0:
            self.tilt_value = self.tilt_value + num
            self.tilt.value = self.tilt_value
        print(self.tilt_value)

    def panIncrement(self, num):
        if self.pan_value < 0.95 and self.pan_value > -0.95:
            self.pan_value = self.pan_value + num
            self.pan.value = self.pan_value
        elif self.pan_value > 0.95 and num < 0:
            self.pan_value = self.pan_value + num
            self.pan.value = self.pan_value
        elif self.pan_value < -0.95 and num > 0:
            self.pan_value = self.pan_value + num
            self.pan.value = self.pan_value
        print(self.pan_value)

cameraAngle = CameraAngle()

class Accelerometer:

    def __init__(self):
        self.i2c = busio.I2C(board.SCL, board.SDA)
        time.sleep(0.2)
        self.accelSensor = adafruit_lsm303_accel.LSM303_Accel(self.i2c)

    def readAccData(self):
        accX, accY, accZ = self.accelSensor.acceleration
        return (accX, accY, accZ)

class ADC:

    def __init__(self):
        self.spi = busio.SPI(clock=board.SCK, MISO=board.MISO, MOSI=board.MOSI) # create the
spi bus
        self.cs = digitalio.DigitalInOut(board.D8) # create the cs (chip select)
        self.mcp = MCP.MCP3002(self.spi, self.cs) # create the mcp object

        self.chanTemp = AnalogIn(self.mcp, MCP.P0)# create an analog input channel on pin 0 for
temperature
```

```
        self.chanSound = AnalogIn(self.mcp, MCP.P1)# create an analog input channel on pin 1 for
sound

    def readSoundData(self):
        self.sound = self.chanSound.value
        return (self.sound)

    def readTemperatureData(self):
        self.temp = self.chanTemp.value
        return self.temp

# Closes relevant processes and stops GPIO
def exit():
    GPIO.output(pwm, GPIO.LOW)
    GPIO.cleanup()
    quit()



def stopGpio():
    GPIO.setmode(gpioMode)
    pixels.fill((0, 0, 0))
    GPIO.output(motorEnL, GPIO.LOW)
    GPIO.output(motorEnR, GPIO.LOW)
    GPIO.output(pwmEnL, GPIO.LOW)
    GPIO.output(pwmEnR, GPIO.LOW)
    pwmEnL.stop()
    pwmEnR.stop()
    GPIO.cleanup()

# Opens the camera preview on the screen
#   Note: for VNC users to see the feed, the setting "Enable Direct Capture Mode" must be on
def start_camera(previewWindow=(452,366, 1380, 715), resolution=(1380,715), rotation=0,
framerate=15):
    camera.preview_fullscreen=False
    camera.preview_window=previewWindow
    camera.framerate = framerate
    camera.resolution=resolution
    camera.rotation = rotation
    camera.start_preview()

def captureImage(fileName):
    try:
        cameraMutex.acquire()
        camera.capture(fileName)
        cameraMutex.release()
    except Exception as e:
        print(f"Error capturing camera image: {e}")

# Closes camera
def stop_camera():
    camera.stop_preview()
    camera.close()


def moveForwardPress(event):
    print(f"Moving forward! Press - Speed = {speed}")
    GPIO.output(motorIn1L, GPIO.HIGH)
    GPIO.output(motorIn2L, GPIO.LOW)
    GPIO.output(motorIn1R, GPIO.LOW)
    GPIO.output(motorIn2R, GPIO.HIGH)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed)


def moveForwardRightPress(event):
    print(f"Moving forward-right! Press - Speed = {speed}")
    GPIO.output(motorIn1L, GPIO.HIGH)
    GPIO.output(motorIn2L, GPIO.LOW)
    GPIO.output(motorIn1R, GPIO.LOW)
    GPIO.output(motorIn2R, GPIO.HIGH)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed/3)


def moveForwardLeftPress(event):
    print(f"Moving forward-left! Press - Speed = {speed}")
    GPIO.output(motorIn1L, GPIO.HIGH)
    GPIO.output(motorIn2L, GPIO.LOW)
    GPIO.output(motorIn1R, GPIO.LOW)
    GPIO.output(motorIn2R, GPIO.HIGH)
    pwmEnL.ChangeDutyCycle(speed/3)
    pwmEnR.ChangeDutyCycle(speed)


def moveBackwardPress(event):
    print(f"Moving backward! Press - Speed = {speed}")
    GPIO.output(motorIn1L, GPIO.LOW)
    GPIO.output(motorIn2L, GPIO.HIGH)
```

```python
    GPIO.output(motorIn1R, GPIO.HIGH)
    GPIO.output(motorIn2R, GPIO.LOW)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed)


def moveBackwardRightPress(event):
    print(f"Moving backward-right! Press - Speed = {speed}")
    GPIO.output(motorIn1L, GPIO.LOW)
    GPIO.output(motorIn2L, GPIO.HIGH)
    GPIO.output(motorIn1R, GPIO.HIGH)
    GPIO.output(motorIn2R, GPIO.LOW)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed/3)


def moveBackwardLeftPress(event):
    print(f"Moving backward-left! Press - Speed = {speed}")
    GPIO.output(motorIn1L, GPIO.LOW)
    GPIO.output(motorIn2L, GPIO.HIGH)
    GPIO.output(motorIn1R, GPIO.HIGH)
    GPIO.output(motorIn2R, GPIO.LOW)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed/3)


def moveLeftPress(event):
    print(f"Turn left! Press")
    GPIO.output(motorIn1L, GPIO.LOW)
    GPIO.output(motorIn2L, GPIO.HIGH)
    GPIO.output(motorIn1R, GPIO.LOW)
    GPIO.output(motorIn2R, GPIO.HIGH)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed)


def moveRightPress(event):
    print(f"Turn right! Press")
    GPIO.output(motorIn1L, GPIO.HIGH)
    GPIO.output(motorIn2L, GPIO.LOW)
    GPIO.output(motorIn1R, GPIO.HIGH)
    GPIO.output(motorIn2R, GPIO.LOW)
    pwmEnL.ChangeDutyCycle(speed)
    pwmEnR.ChangeDutyCycle(speed)
```

```
def moveRelease(event):
    print(f"Release movement button")
    pwmEnL.ChangeDutyCycle(0)
    pwmEnR.ChangeDutyCycle(0)

def stopMovement():
    print(f"Emergency stop!")
    GPIO.output(motorIn1L, GPIO.LOW)
    GPIO.output(motorIn2L, GPIO.LOW)
    GPIO.output(motorIn2R, GPIO.LOW)
    GPIO.output(motorIn1R, GPIO.LOW)
    pwmEnL.ChangeDutyCycle(0)
    pwmEnR.ChangeDutyCycle(0)

def lock():
    print(f"Locking suspension")

def ledOn():
    print(f"Turning on LED")
    pixels.fill((255, 255, 255))

def ledOff():
    print(f"Turning off LED")
    pixels.fill((0, 0, 0))

# Testing purposes only
def motorTurnTest():
    print(f"Testing DC motor")
    print(f"What goes up...")
    for dc in range(0, 101, 2):
        #motor.setVelo(dc)
        motors.go(dc)
        time.sleep(0.05)
    time.sleep(1)
    print(f"...must come down")
    for dc in range(100, -1, -2):
        #motor.setVelo(dc)
        motors.go(dc)
        time.sleep(0.05)
    time.sleep(1)
    print(f"Aaaand backwards")
    for dc in range(0, -101, -2):
        #motor.setVelo(dc)
        motors.go(dc)
        time.sleep(0.05)
    print(f"And back")
    for dc in range(-100, 1, 2):
        #motor.setVelo(dc)
        motors.go(dc)
        time.sleep(0.05)
    print(f"Motor turn done")


def cameraUp():
    cameraAngle.tiltIncrement(-0.1)


def cameraDown():
    cameraAngle.tiltIncrement(0.1)


def cameraLeft():
    cameraAngle.panIncrement(0.1)


def cameraRight():
    cameraAngle.panIncrement(-0.1)
```